

Nucleus NET

Reference Manual

0001008-002 Rev. 101



Copyright (c) 2000
Accelerated Technology, Inc.
720 Oak Circle Dr. E.
Mobile, AL 36609
(334) 661-5770



Related Documentation

Nucleus PLUS Reference Manual, by Accelerated Technology, describes the operation and usage of the Nucleus PLUS kernel.

Nucleus PLUS Internals, by Accelerated Technology, describes, in considerable detail, the implementation of the Nucleus PLUS kernel.

Style and Symbol Conventions

Program listings, program examples, filenames, menu items/buttons and interactive displays are each shown in a special font.

Program listings and program examples - *Courier New*

Filenames - *COURIER NEW, ALL CAPS*

Interactive Command Lines - ***Courier New, Bold***

Menu Items/Buttons – *Times New Roman Italic*

Trademarks

MS-DOS is a trademark of Microsoft Corporation

UNIX is a trademark of X/Open

IBM PC is a trademark of International Business Machines, Inc.

Additional Assistance

For additional assistance, please contact us at the following:

Accelerated Technology

720 Oak Circle Drive, East

Mobile, AL 36609

800-468-6853

334-661-5770

334-661-5788 (fax)

support@atinucleus.com

<http://www.atinucleus.com>

Copyright (©) 2000, All Rights Reserved.

Document Part Number: 0001008-002 Rev.101

Last Revised: January 27, 2000





Contents

Chapter 1 - Introduction	1
Networking In General	2
Transport Layer.....	3
Transmission Control Protocol (TCP)	3
User Datagram Protocol (UDP)	3
Network Layer	4
Raw Internet Protocol (Raw IP).....	4
Data Link Layer	5
Physical Layer.....	5
Chapter 2 – Getting Started	7
Application Development Overview	8
Installing Nucleus NET.....	8
Network Initialization	9
Transferring Data Using Nucleus NET.....	9
Building an Address Structure	9
Standard Socket-like Usage	10
Standard UDP Usage	11
Raw IP Usage.....	12
Configuration Options	12
Network Configuration	12
Local Host Adapters Definition (NU_DEVICES)	13
Host Processor Environment Configuration	15
Chapter 3 – Functional Description.....	17
Nucleus NET Initialization	18
TCP Interface	19
Server.....	19
Client.....	19
Sending Data.....	20
Receiving Data.....	20
UDP Interface	21
Server.....	21
Client.....	21



Raw IP Interface	22
Server	22
Client.....	22
Chapter 4 - Nucleus NET Services.....	23
NU_Abort	25
NU_Accept	26
NU_Add_DNS_Server.....	28
NU_Add_Route	29
NU_Bind.....	30
NU_Bootp.....	31
NU_Close_Socket.....	36
NU_Connect	37
NU_Delete_DNS_Server	39
NU_Dhcp	40
NU_Dhcp_Release.....	43
NU_Fcntl.....	45
NU_FD_Check.....	46
NU_FD_Init	47
NU_FD_Reset.....	48
NU_FD_Set.....	49
NU_Get_DNS_Servers	50
NU_Get_Host_By_Addr.....	51
NU_Get_Host_By_Name	52
NU_Get_Peer_Name	53
NU_Getsockopt.....	54
NU_Ioctl	56
NU_Init_Devices	58
NU_Init_Net	59
NU_Is_Connected.....	61
NU_Listen.....	62
NU_Ping	63
NU_Rarp.....	64
NU_Recv.....	66
NU_Recv_From.....	67
NU_Recv_From_Raw.....	69
NU_Rip2_Initialize	71
NU_Select	73
NU_Send.....	75
NU_Send_To	77
NU_Send_To_Raw	79
NU_Setsockopt	81
NU_Socket.....	83



Chapter 5 – Extended Discussion.....	85
Blocking on Read Service.....	86
Nucleus NET Task Priorities	86
Including SNMP Support.....	87
Domain Name System (DNS).....	87
IP Multicasting.....	88
DHCP (Dynamic Host Configuration Protocol)	89
Including Routing Information Protocol (RIP2) Support.....	93
Chapter 6 – Using Nucleus NET With Nucleus PLUS	95
Tasks	96
Control/Coordination Task	96
Timer Notification Task.....	97
Nucleus PLUS Semaphore.....	98
Nucleus PLUS Queue	98
Chapter 7 – Driver Assistance Functions	99
Introduction.....	100
“MII_AutoNeg” (MII Automatic Negotiate)	100
Appendix A –Values Returned By Socket Interface	103
Appendix B – Sample Application.....	107
Appendix C – Changes to the Nucleus API	115





1

Introduction

Networking in General

Transport Layer

Network Layer

Data Link Layer

Physical Layer



Nucleus NET is a networking extension for the Nucleus PLUS real-time kernel. Its purpose is to provide interconnection with well-known host system environments, and to facilitate communications between loosely coupled nodes within the user's proprietary system.

Nucleus NET, developed based on the NCSA implementation, has been thoroughly reviewed and modified to promote better compliance with the TCP/IP networking standard, and to improve efficiency for embedded applications.

In order to provide both compatibility with existing standards and interfaces, while at the same time providing a means to operate the network stacks efficiently, Accelerated Technology provides a sockets API that is modeled after Berkley Sockets. Where efficiency would suffer the Nucleus NET sockets API deviates from the Berkley Sockets.

Nucleus NET, wherever possible, hides the details of the actual network implementations. However, in some cases, for efficiency purposes, the user sees some of the internal data structures used in the Nucleus NET protocol stack.

Networking In General

Networking continues to grow in popularity with users of desktop, mini, and mainframe computers. Consequently, embedded developers are adding networking support to their systems. Unfortunately, most networking implementations available today are based on the UNIX model provided by both BSD 4.3 and ATT System 5. This is fine for compatibility purposes, but efficiency of execution and memory usage must be sacrificed. Nucleus NET is designed to be a 'living' software component. That is, Accelerated Technology will continually strive to keep up with the latest standards and maintain the highest level of compliance possible where it makes sense. In this light, Accelerated Technology's efforts toward improving the compliance of their networking software with current RFCs will be balanced with the need for efficient implementation in terms of both size and execution time.

In general, networking software is designed using a layered approach. Each software component within the networking 'stack' should provide a separate and distinct interface and semantics from the adjacent later in the stack. This provides for clear delineation of interpretable software components. The Application Layer, Presentation Layer and Session Layer are not currently supported by the Nucleus NET code. For example, the transport layer is expected to communicate with the transport layer on another network node. The same is true for the Networking layer, and so on. An exhaustive discussion of the layers and their purposes is beyond the scope of this document.

A short description of each layer used by Nucleus NET follows. As mentioned above, a complete discussion would be out of scope, and it is questionable as to whether it would actually aid the developer in using Nucleus NET. However, the following descriptions will lend understanding to the use of Nucleus NET. They will be kept brief and to the point so that we are not bogged down in the details of network theory.



Transport Layer

This layer is the highest-level protocol layer provided to Nucleus NET users (an exception to this occurs when the user accesses add-on products such as FTP, TELNET, etc.). Its purpose is to provide an interface that hides the details of the actual implementation of the protocol stack. The transport layer provides both connection oriented and connectionless access to the network. The connection-oriented protocol provided is the Transmission Control Protocol (TCP). Connectionless network access is provided to the user by the User Datagram Protocol (UDP).

Transmission Control Protocol (TCP)

The TCP protocol is provided for those users who require a high degree of confidence that their network traffic is being accomplished reliably. Other than internal mechanisms to the protocol stack, which ensure reliability, the user is required to make a 'connection' over TCP before data can be transferred in either direction. A connection is similar to that encountered in a typical telephone call. One individual calls another, the phone rings, the called individual answers, the caller hears the answer and acknowledges it, and the connection is established.

Connections on TCP take place within the context of a Client/Server relationship. In the telephone example, the individual receiving the call is analogous to the server. In general (this is where the analogy breaks down) a server is doing just that, providing a service (e.g., printing, file access, time, etc.). A server performs a series of actions that place it in a condition to receive 'connection requests' by 'listening' on a well-known 'port' address, or 16-bit address, and does not continue processing activity until a connection is established. Meanwhile, the client sends a request across the network to connect to the well-known port. Once the server recognizes this request the internals of the TCP/IP stack negotiate the connection. Once the connection is established, the TCP/IP stack informs the server and the client on their respective hosts.

Connections are fully described by two 'sockets'. A socket is an IP address/port number pair (see the Network Layer section of this chapter for an explanation of the IP makeup). A socket pair uniquely identifies a TCP connection on a network. That is, the server's port/IP address combination and the client's port/IP address combination form a pair of sockets.

Data transfer over TCP is straightforward. Either the server or the client can send or receive data at will. The TCP protocol is a full duplex circuit. That is, data can be transferred in both directions simultaneously.

Once communications have been completed (both hosts are finished communicating) the connection is closed and its associated resources are freed for other server/client usage.

User Datagram Protocol (UDP)

UDP is used for high-speed data transfer between two nodes on a network. A modified, less formal, client/server relationship is established between two nodes employing UDP. No connection is required for data transfer. One node decides to send a message to another node. The system must be set up such that the recipient of the data is aware of the data transmission. The recipient is therefore 'waiting' for a message over the network.



Again, port numbers are used. The sender sends the message to a specified port/IP number pair.

Data transfer continues at will, as long as there is a node on the network that is willing to receive the data.

Network Layer

In general, this layer is somewhat transparent to the user of the network. Accelerated Technology provides a raw interface to this layer for those applications that wish to use the network without a transport layer. The purpose of the network layer is primarily to provide an 'inter-network' capability. A network is generally considered a physically connected set of nodes communicating on the same electronic medium (e.g., ethernet). In order to interconnect one physical network to another, some mechanism is needed to provide translation between the possibilities of differing physical network characteristics (e.g., one ethernet network to a token ring network, or one ethernet network to another ethernet network). These inter-networking facilities are referred to as 'gateways', that is, they provide passage from one physical network to another. This requires two distinct hardware facilities on one node so that traffic can be taken from one and transferred to another.

In Nucleus NET, network layer services are provided by the Internet Protocol (IP) layer. Individual IP nodes on a network are specified by a 32 bit address, where it is subdivided into four 8 bit values, each ranging from 0 - 255. This address is always displayed in the 'Dot Decimal' formation, i.e. '192.4.100.0'. Each IP address is divided into several classes, where the first five bits of the address are used to define the class. Class A addresses have the first bit as a '0', Class B addresses have '1 0' as the first two bits, Class C addresses have '1 1 0' as the first three bits, Class D addresses have '1 1 1 0' as the first four bits, and Class E addresses have '1 1 1 1 0' as the first five bits. The fourth byte designates the physical network that a node is attached to.

Raw Internet Protocol (Raw IP)

The Raw IP interface is used by applications that wish to directly access the network. Two features that UDP and TCP do not provide are provided by the Raw IP interface. A Raw IP interface can read and write Internet Protocol datagrams with a protocol field of 255 (Raw Protocol), 63 (Hello Protocol), 89 (OSPF Protocol), or 0 (WildCard Protocol). For example, the OSPF routing protocol does not use TCP or UDP, but uses IP directly, setting the protocol field of the IP datagram to 89. Also, an application that uses a Raw IP interface can build its own IP datagram header, using the `IP_HDRINCL` socket option.

A client/server relationship is established between two nodes employing the Raw IP interface. No connection is required for data transfer. One node decides to send a message to another node. The system must be set up such that the recipient of the data is aware of the data transmission. The recipient is therefore 'waiting' for a message of a particular protocol over the network. The sender sends a message of a particular protocol to a specified IP number pair. (IP numbers are explained in succeeding paragraphs.) There are several criteria that determine if the recipient of a Raw IP interface will accept an incoming datagram. If the recipient is waiting for a datagram of a nonzero protocol, then the received datagram's protocol field must match the nonzero protocol that the recipient is waiting for, or the datagram is not delivered to the socket. If a local IP



address is bound to the Raw IP socket, then the destination IP address of the received datagram must match this bound address, or the datagram is not delivered to the recipient. If a foreign IP address was specified by the recipient, then the source IP address of the received datagram must match the connected address, or the datagram is not delivered to the recipient. If the recipient is waiting for a datagram of zero protocol, then the received datagram's protocol field can be any nonzero protocol to be accepted. Notice that if a Raw IP socket is created with a protocol of 0, and the local or foreign address is not specified then the socket will receive a copy of every incoming raw datagram.

Data Link Layer

The data link layer provides functional and procedural means to establish, maintain, and release data links between networks. This layer handles the error-free data transmission on noisy lines, error correction, and possible retransmission of data if it was sent poorly. This layer may use a checksum type value added to the message data for error correction.

Physical Layer

The physical network layer controls access to the physical medium by which the network is connected. In most cases for Nucleus NET the medium is the ethernet standard. However, the medium can be replaced for other types such as token ring or ARCnet. Differences between physical network connections (other than the electrical characteristics) are embodied in the addressing mechanism. For example, ethernet uses a 48-bit ethernet address to uniquely identify one node on the network from another.

The physical layer interfaces directly with the data link layer. The Nucleus NET file "NET.C" is where all the code is placed for this interface control of the hardware. All the functions for "*Sending*", "*Receiving*", "*Flow Control*", etc. are in this file.





Getting Started

Application Development
Overview

Installing Nucleus NET

Network Initialization

Transferring Data Using Nucleus
NET

Building an Address Structure

Standard Socket-like Usage

Standard UDP Usage

Raw IP Usage

Configuration Options

Network Configurations

Local Host Adapters Definition

Host Processor Environment
Configuration

The following sections describe the use of Nucleus NET in general. Nucleus NET provides a series of enhancements over the commonly available interfaces to typical TCP/IP networking software implementations. These enhancements provide for more compact network processing. By providing memory allocation facilities, Nucleus NET can greatly reduce the amount of time spent processing network requests on an embedded host. However, the overall throughput of the network traffic is dictated by the slowest host involved in the network communication (notwithstanding the network traffic in general).

Application Development Overview

Nucleus NET applications are written within the context of a Nucleus PLUS system. This means that full multi-tasking mechanisms are expected and required. Nucleus NET itself uses two tasks. One task is responsible for processing all events generated by the stack. An example of such an event would be the retransmission of a TCP packet. The second task processes received packets. See Chapter 6 for more on the configuration of the Nucleus PLUS setup.

Users of the Nucleus NET protocol stack access network services from a task. That is, when a Nucleus NET service is requested, it executes in the context of the requesting task.

Nucleus NET provides for both blocking and non-blocking services. Some services block by default, others do not. If non-blocking services are required, Nucleus NET provides the `NU_Fcntl` service. Non-blocking services return immediately whether the request is completed or not. Therefore, it is possible in a non-blocking read request that no bytes will be returned.

Write services only block when there are not enough resources (buffer space) available to send the data. The task is resumed when resources become available and the write request can be satisfied. Note that write services always block when resources are not available. The `NU_Fcntl` cannot be used to make a write request non-blocking.

Active connection requests are always blocked. A client performs an active connection request. Passive connection requests made by a TCP server can be either blocking or non-blocking. Except for the passive connection request of a TCP server, control will never return if a connection is not established. When a write request is made, the task may assume that the data will be transferred to the specified recipient (unless a parameter error was detected by the protocol stack, e.g., an incorrect socket identifier was specified).

Details describing the use of the various Nucleus NET capabilities are described in Chapter 3, Functional Description and Chapter 4, Nucleus NET Services.

Installing Nucleus NET

The installation of Nucleus NET varies with the target platform. Please refer to the *Target Specific Notes* that you received with Nucleus NET for complete installation instructions.



Network Initialization

Before using any of the network services the stack and the physical layer devices (ethernet controllers, UARTS, etc.) must be initialized. Network Initialization should be performed from within a task, i.e., not in the Nucleus PLUS `Application_Initialize` routine. Preferably, network initialization should be performed in the first task to execute. All network applications must call `NU_Init_Net` and `NU_Init_Devices`. Additionally if a host is going to discover it's IP address dynamically it may have to call one or more of the following functions: `NU_Rarp`, `NU_Dhcp`, or `NU_Bootp`. All of the aforementioned function calls should occur after the `NU_Init_Devices` call. When Nucleus NET initialization is requested, data structures for the protocol stack are initialized and control returns to the requestor. Nucleus NET initialization is performed by simply calling `NU_Init_Net`.

Nucleus NET handles multiple network interfaces and before calling `NU_Init_Devices` the `NU_Device` structure should be filled out for each device that is to be used in the physical layer. Once the device structure is filled with the appropriate values, the `NU_Init_Devices` function is called. This function initializes the data structures, adds each device to the list of devices, calls the devices initialization function, and attaches IP addresses to the device. Data structures for connections, data transfers, etc., are not allocated until subsequent connection, read, and write requests are made.

Transferring Data Using Nucleus NET

Nucleus NET can be used in a number of ways. A TCP connection or connections can be made and data transferred in the standard socket-like manner permitted by Nucleus NET. The standard UDP interface can be used, or the Raw IP interface can be used.

In the following paragraphs you will note that in order to begin communications you must "open a socket". This process is preceded by building an address structure to specify the address of the node you wish to connect or send data to, and an address indicating your node on the network (in certain cases). The next paragraph describes the data structures and process necessary to build an address.

Building an Address Structure

An address structure is built using the predefined structure `addr_struct`. The `addr_struct` contains information about a node on the network including its family type, port number, IP number, and name.



The following code fragment describes how an address structure would be built for an IP node with a port of 4000, an IP address of 100.200.40.1 and a name of `test_node`.

```
/* Declare the address structure. */
struct addr_struct *test_addr;

/* Allocate memory for the address structure. */
status = NU_Allocate_Memory (&System_Memory, (void *)&test_addr,
                             sizeof(struct addr_struct),
                             NU_SUSPEND);

/* Indicate this is an IP family address. */
test_addr->family = NU_FAMILY_IP;

/* Set up the port number. */
test_addr->port = 4000;

/* Set up the IP number. */
test_addr->id.is_ip_addrs[0] = (unsigned char) 100;
test_addr->id.is_ip_addrs[1] = (unsigned char) 200;
test_addr->id.is_ip_addrs[2] = (unsigned char) 40;
test_addr->id.is_ip_addrs[3] = (unsigned char) 1;

/* Set up the name. */
test_addr->name = "test_node";
```

The address specified depends on the function being performed. In general, when you are defining a server node (either using UDP or TCP) you will build an address structure for yourself. If you are performing a connect to a TCP port or sending data on a UDP port, you will build an address structure with the information pertaining to the node you are connecting or sending data to. One exception to this is when you are acting as a server for a UDP port. In this case, if you perform an `NU_Recv_From` call prior to the `NU_Send_To` call, the socket library will build the “to” address information for you. In this case, if you wish to send data to a node other than the node you received data from, you simply build an address structure before you send data to it.

Standard Socket-like Usage

As mentioned in the previous section, TCP communications require that the user choose whether to act as a server or client. Once that choice has been made, follow the steps outlined below:



Server

- Create a socket (NU_Socket)
- Bind the Server port and IP address (NU_Bind)
- Perform a Listen call to establish the number of simultaneous requests that can be queued (NU_Listen)
- Perform an Accept service call to “wait” for client connections (NU_Accept)
- After the Accept service call returns (i.e., a connection has been established) data can be transferred (control returns from NU_Accept)
- Transfer data (NU_Send and NU_Recv)
- Close the connection (NU_Close)

Client

- Create a socket (NU_Socket)
- Perform a Connect service call (NU_Connect)
- After the Connect service call returns (i.e., a connection has been established) data can be transferred (control returns from NU_Connect)
- Transfer data (NU_Send and NU_Recv)
- Close the connection (NU_Close)

Standard UDP Usage

UDP is inherently connectionless. However, it is expected that in general, both sides of a data transfer are aware of the transfer, and port numbers have been predefined. The operation for a sender and receiver only differ in the service calls that are made. That is, both the sending and receiving of data require a full address specification of the recipient and sender respectively. Data transfer over UDP is simply performed by using the appropriate interfaces specified in Chapter 4.

UDP can operate in both server and client mode. Additionally, it can be used to simply communicate from one machine to another without the notion of a server. However, in order to receive a Datagram, a port must be specified, and the application must have issued a NU_Recv_From service call before the sender actually sends the packet in order for the UDP to process the Datagram.

The steps involved in transferring data via UDP are:

Server

- Create a socket (NU_Socket)
- Bind the local server address to the socket (NU_Bind)
- Wait on client’s data (NU_Recv_From)
- Send data — optional (NU_Send_To)
- Close the socket (NU_Close)



Client

- Create a socket (`NU_Socket`)
- Send data to the server (`NU_Send_To`)
- Receive data — optional (`NU_Recv_From`)
- Close the socket (`NU_Close`)

Raw IP Usage

Raw IP is connectionless. However, it is expected that in general, both sides of a data transfer are aware of the transfer, and that the protocol being used has been established. The operation for a sender and receiver only differ in the service calls that are made. Data transfer over Raw IP is simply performed by using the appropriate interfaces specified in Chapter 4.

Raw IP can operate in both server and client mode. Additionally, it can be used to simply communicate from one machine to another without the notion of a server. However, in order to receive a Datagram, a protocol must be specified, and the application must have issued a `NU_Recv_From_Raw` service call before the sender actually sends the packet in order for Raw IP to process the datagram.

The necessary steps involved in transferring data via Raw IP are:

Server

- Create a socket (`NU_Socket`)
- Wait on client's data (`NU_Recv_From_Raw`)
- Send data — optional (`NU_Send_To_Raw`)
- Close the socket (`NU_Close_Socket`)

Client

- Create a socket (`NU_Socket`)
- Send data to the server (`NU_Send_To_Raw`)
- Receive data — optional (`NU_Recv_From_Raw`)
- Close the socket (`NU_Close_Socket`)

Configuration Options

Target environments as well as the TCP/IP protocol stack require certain configuration information in order to function properly. Configuration options are conveniently isolated in separate include files for easy modification. The files and their associated configuration items are provided in the following section.

Network Configuration

The structure `NU_DEVICES`, is used to add multiple interfaces into the Nucleus NET physical Layer. Parameters for setting up the Nucleus NET interfaces are specified in the structure `NU_DEVICES`. The following is a discussion of this data structure.



Local Host Adapters Definition (NU_DEVICES)

Structure `NU_DEVICES` specifies the setup parameters for the Nucleus NET host adapters. Information in this table defines hardware usage, names, IP numbers, and data transmission requirements (note that pointers are shown as 16 bits, this of course is target dependent and is only shown here to simplify the figure).

CHAR *dv_name	
INT32 dv_flags	
UINT32 dv_driver_options	
STATUS (*dv_init) (DV_DEVICE_ENTRY *)	
UINT8 dv_ip_addr[0]	UINT8 dv_ip_addr[1]
UINT8 dv_ip_addr[2]	UINT8 dv_ip_addr[3]
UINT8 dv_subnet_mask[0]	UINT8 dv_subnet_mask[1]
UINT8 dv_subnet_mask[2]	UINT8 dv_subnet_mask[3]
UINT8 dv_gw[0]	UINT8 dv_gw[1]
UINT8 dv_gw[2]	UINT8 dv_gw[3]
INT32 dv_hw.udp.com_port	
INT32 dv_hw.udp.baud_rate	
INT32 dv_hw.udp.parity	
INT32 dv_hw.udp.stop_bits	
INT32 dv_hw.udp.data_bits	
UINT32 dv_hw.ether.dv_irq	
UINT32 dv_hw.ether.dv_io_addr	
UINT32 dv_hw.ether.dv_shared_addr	



A description of each of the fields for the DEV_DEVICES structure follows:

Field	Description
dv_name	Unique device name, that is used to look up the interface.
dv_flags	Any special flags used for configuring the operation of the device, ex. DV_NOARP.
(*dv_init) (DV_DEVICE_ENTRY *)	The name of the interface's initialization routine.
dv_driver_options	Any driver dependent options, including the possibility of the address of a structure containing driver options
dv_ip_addr	The IP number to be used by this Nucleus NET interface.
dv_subnet_mask	Four characters representing the subnet mask being used by this Nucleus NET interface.
dv_gw	Four characters representing the gateway to be used for this interface.
dv_hw.uart.com_port	The COM Port to be used for this interface if the interface is an uart.
dv_hw.uart.baud_rate	The Baud Rate to be used for this interface if the interface is an uart.
dv_hw.uart.data_mode	The Data Mode to be used for this interface if the interface is an uart.
dv_hw.uart.parity	The Parity to be used for this interface if the interface is an uart.
dv_hw.uart.stop_bits	The Stop Bits to be used for this interface if the interface is an uart.
dv_hw.uart.data_bits	The Data Bits to be used for this interface if the interface is an uart.
dv_hw.ether.dv_irq	The Irq to be used for this interface if the interface is an ethernet controller.
dv_hw.ether.dv_io_addr	The I/O address to be used for this interface if the interface is an ethernet controller.
dv_hw.ether.dv_shared_addr	The Sharer Memory address to be used for this interface if the interface is an ethernet controller.



Host Processor Environment Configuration

All network traffic, other than data that is encapsulated, must be presented in network byte order. Network byte order is big endian, that is, the least significant byte is transferred first in multi-byte data elements. To facilitate processors which are inherently little endian, certain routines are provided that convert data from the host format to network byte order. These are `INTSWAP` and `LONGSWAP`. These functions are implemented such that they are architecture independent. When invoked on a big endian architecture they have no effect, i.e., the byte ordering is not changed.

In addition to network byte ordering, different machines, and consequently compilers, use different size integers. The natural integer assumed by Nucleus NET is 16 bits. Therefore, special `#define` statements have been inserted into the software to facilitate conversion from a compiler's natural integer to 16-bit integers. The file `TARGET.H` should be examined to determine that the types defined therein are accurate for the target processor environment. This file should have been sent with your port, and therefore should already be set up for your compiler and machine type.

The number of sockets, TCP ports, UDP ports, and Raw IP sockets must also be specified or the default values will be used. The following is a list of definitions associated with these values and the files in which they can be found for modification.

Define	Default	File	Description
NPORTS	30	TARGET.H	Number of TCP ports
NUPORTS	30	TARGET.H	Number of UDP ports
NIPORTS	30	TARGET.H	Number of Raw IP sockets

Unallocated ports and sockets are only one integer in length. Therefore, it does not consume too much overhead to provide for more.





3

Functional Description

Nucleus NET Initialization

TCP Interface

UDP Interface

Raw IP Interface



As mentioned previously, Nucleus NET is based on the TCP/IP protocol standard. The user of Nucleus NET can utilize a series of interfaces referred to as a socket library. The socket library provides interfaces to the networking initialization, connection oriented data transfer capabilities, connectionless data transfer capabilities, and a high-speed interface referred to as a Raw IP Interface.

The connection oriented interface provides access to the Transmission Control Protocol (TCP) capabilities of Nucleus NET. The connectionless interface provides access to the User Datagram Protocol (UDP) capabilities of Nucleus Net. The Raw IP Interface provides direct access to the data transfer and consequently, addressing capabilities of the Internet Protocol services provided by Nucleus Net.

Within the context of these interfaces, the user often has a choice of the paradigm he will use. In general, the user will choose between operations as a server or as a client.

A general functional description on the variations of these interfaces is provided in this section. It is meant as a brief overview to familiarize the reader with some of the basic operations of Nucleus NET.

Nucleus NET Initialization

Nucleus NET initialization must be performed before any networking services are available to the user. The initialization of the network is activated by calling the routine `NU_Init_Net`.

Initialization of the network consists of configuring the initializing global data structures for timing and port maintenance, setting up the driver interface, initializing the hardware, and initializing the packet header templates for the physical (e.g., ethernet) layer, the IP layer, the TCP layer, the UDP layer, ARP, and ICMP protocols.

Once initialization is complete, the remaining network services can be accessed. An invalid initialization is indicated by a negative value returned by `NU_Init_Net`.



TCP Interface

TCP, the most commonly used interface of Nucleus NET, accesses some fairly complex facilities. TCP is responsible for ensuring reliable communications and therefore maintains facilities that account for packets sent that are acknowledged and acknowledging packets received from another node. To ensure reliability further, TCP performs extensive checksumming operations.

From a user interface perspective, TCP assists in establishing connections, sending data, and receiving data. The TCP interface is generally used in a client/server environment. The following paragraphs describe these modes.

Server

When setting up the request to act as a server, the TCP constructs a socket entry for the requester, `NU_Socket` that essentially initializes the socket to an empty state.

The TCP then is instructed to “bind” a local address structure to the socket. The local address structure is defined by the user prior to the `NU_Bind` service call.

Once the socket and local address information have been “bound” (copied from the local structure into the socket structure) the user specifies to the TCP how many concurrent connection attempts it will accept while processing an ongoing request `NU_Listen`. This request constructs an internal table used by the TCP to determine if an incoming connection request can be processed.

Client

As with the server, the client task first requests a socket to be allocated for its TCP communication by invoking the `NU_Socket` service call.

Once a socket has been allocated, the client task builds an address structure for the host to which it intends to connect. The information for the host can be defined in the `defmachinfo` table to reduce the overhead in `NU_Connect`. (It doesn't have to allocate and build an entry for the request.) Or, when the proper values are provided to `NU_Connect`, it will construct a machine table for the request (the IP number and the name should be defined as the server address structure passed to `NU_Connect`, see the section on `NU_Connect` in Chapter 4).

The `NU_Connect` service call suspends the requesting task until a connection is completed. Once the connection is completed, control returns to the requesting task and it can begin transferring data.



Sending Data

Transferring data is identical for both server and client tasks. When sending data via the `NU_Send` service call, the user specifies the socket descriptor that pertains to the connection over which data is to be transmitted along with other parameters.

`NU_Send` verifies that there is a valid connection for the request and copies the data from the requester's buffer into the internal TCP output buffer. If there are no TCP output buffers available to store the data then the task will be suspended until buffers are available. Once the data has been copied, the TCP builds a header for the packet and sends it to the receiving node. Control returns immediately to the requester once the data has been copied into the TCP structures. The requester should assume that the data is properly sent to the recipient and that the recipient received it in a valid condition. The exception to satisfactory data transmission and reception is when the network is broken. If the network is broken before the request is made, an error condition is returned from the `NU_Send` request. If this occurs, the user should cease using the socket descriptor associated with the send.

If the connection breaks after the `NU_Send` service call returns but before the data is successfully transmitted and received, subsequent calls to the socket interface will return an error condition and the sending task should assume that the last send request was not completed successfully.

Receiving Data

When receiving data via the `NU_Recv` service call, the user specifies the socket descriptor that pertains to the connection from which data is to be received along with other parameters.

`NU_Recv` verifies that there is a valid connection for the request and then makes a decision based on whether the socket being used is currently set up for blocking or non-blocking receives.

If the requester is set up for blocking on receive, `NU_Recv` makes sure that there is no data waiting on the connection. If there is data, it is returned to the requester. Otherwise, the requester is suspended until data is successfully received. When the data is received, `NU_Recv` returns to the requester the number of bytes actually received.

If the requester is NOT set up for blocking on receive, `NU_Recv` checks if any data is waiting on the connection and, if so, returns the number of bytes received to the requester. Otherwise the number of bytes returned is 0.

If the network is broken, `NU_Recv` returns an appropriate error code. At that point, the requestor should cease the use of the socket.



UDP Interface

The UDP services provided by Nucleus NET can be used for high speed, low overhead transmission and reception of data over the network. Higher transmission rates and lower overhead are achieved by eliminating the reliability management used by TCP. Hence, UDP is inherently unreliable.

UDP services are used in much the same way as TCP services, that is, sockets are allocated, addresses are specified, servers and clients are generally assumed, and data is transferred. However, since UDP is connectionless, no connections are established or torn down.

Server

When setting up to act as a UDP server, an address structure is defined which specifies the server port number and IP number. After the address structure has been built and a socket has been allocated via `NU_Socket`, the address is bound to the socket via a call to `NU_Bind`.

Once the address has been bound, the UDP server can issue a `NU_Recv_From` service call. UDP is inherently blocking in Nucleus NET; therefore, when `NU_Recv_From` is called, the task making the call is suspended and control will not return to it until data is actually received on the specified UDP port.

Once data has been received, the UDP server can send data back to the node from which it received the data by simply issuing a `NU_Send_To` service call. The socket library will “fill in” the recipient’s address information as long as a client address structure has been allocated and its address is passed to the `NU_Send_To` service call. If the server wishes to send data to a UDP node or port other than the one it received data from, it must build an address structure for that node and specify the recipient’s address structure in the `NU_Send_To` service call.

Once communication as a UDP server is completed, a call to `NU_Close_Socket` should be issued to deallocate the data structures associated with the UDP activity.

Client

When acting as a UDP client, the user must build an address structure for the server node. The address for the local client node will be built automatically by the socket interface. In addition, the user must allocate a socket `NU_Socket` for the communication. Once the address has been specified and the socket has been created, the UDP client can send and receive data via the `NU_Send_To` and `NU_Recv_From` service calls respectively. After communication is completed as a UDP client, `NU_Close_Socket` should be called to deallocate the structures associate. After communication is completed as a UDP client, `NU_Close_Socket` should be called to deallocate the structures associated with the UDP communications.



Raw IP Interface

The Raw IP services provided by Nucleus NET can be used for high speed, low overhead transmission and reception of data over the network. The Raw IP Interface provides direct access to the data transfer and consequently, addressing capabilities of the Internet Protocol services provided by Nucleus Net. Again, like UDP, higher transmission rates and lower overhead are achieved by eliminating the reliability management used by TCP. Hence, Raw IP is inherently unreliable.

Raw IP services are used in much the same way as UDP services, that is, sockets are allocated, servers and clients are generally assumed, and data is transferred. Like UDP, Raw IP is connectionless, no connections are established or torn down.

Server

When setting up to act as a Raw IP server, an address structure is defined which specifies the server port number of zero and an IP number. A socket has been allocated via `NU_Socket` that specifies the protocol desired.

Raw IP is inherently nonblocking in Nucleus NET; therefore, if blocking is required the task must set the socket to blocking by issuing a `NU_Fcntl`. The Raw IP server can then issue a `NU_Recv_From_Raw` service call.

Once data has been received, the Raw IP server can send data back to the node from which it received the data by simply issuing a `NU_Send_To_Raw` service call. The socket library will “fill in” the recipient’s address information as long as a client address structure has been allocated and its address is passed to the `NU_Send_To_Raw` service call. If the server wishes to send data to a Raw IP node other than the one it received data from, it must build an address structure for that node and specify the recipient’s address structure in the `NU_Send_To_Raw` service call.

Once communication as a Raw IP server is completed, a call to `NU_Close_Socket` should be issued to deallocate the data structures associated with the Raw IP activity.

Client

When acting as a Raw IP client, the user must build an address structure for the server node. The address for the local client node will be built automatically by the socket interface. In addition, the user must allocate a socket with `NU_Socket` for the communication. Once the address has been specified and the socket has been created, the Raw IP client can send and receive data via the `NU_Send_To_Raw` and `NU_Recv_From_Raw` service calls respectively.

After communication is completed as a Raw IP client, `NU_Close_Socket` should be called to deallocate the structures associated with the Raw IP communications.



4

Nucleus NET Services



All Nucleus NET access should be contained within the confines of the socket library. However, it is possible for calls to be made to the lower layers. Caution is in order when accessing the protocol stack without using the socket interface, however. This is due to the fact that all access to the protocol stack is semaphore protected to restrict access and preclude reentrancy failures.

Most Nucleus NET services provide return values, which indicate the status of the request. See the following sections for the specific values returned for each of the interfaces. See (Values Returned by Socket Interface, Appendix A) for the values associated with the return codes specified below

The following sections describe the interfaces available to the Nucleus NET user. Each section includes a description, the prototype and meaning of parameters, possible return values, and an example of usage for the interface.



NU_Abort

```
STATUS NU_Abort (socketd);
```

This function is responsible for aborting a TCP or UDP connection. In the case of a TCP connection, a RESET is sent to the remote host. All resources are freed up.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.

Return Value

Code	Meaning
NU_SUCCESS	The connection was successfully aborted.
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.

Example

```
INT socketd;  * the original socket descriptor */

status = NU_Abort(socketd);
/* if status is NU_SUCCESS then the connection was
   successfully aborted */
```



NU_Accept

```
STATUS NU_Accept (socketd, peer, addrlen);
```

This function is responsible for establishing a new socket descriptor containing information on both the server and the client. It is only necessary for the server to call this function when a connection-oriented transfer is being established. `NU_Accept` can be used in either a blocking mode or a non-blocking mode. In blocking mode the calling task will be suspended until a connection is made. In non-blocking mode, the service returns immediately, even if no connections are ready to be accepted. Blocking is the default mode of operation. To make the service non-blocking, the `NU_Fcntl` service must be used first.

IMPORTANT: Prior to the `NU_Accept` service call, an application must call `NU_Listen`. `NU_Listen` sets up a table to accept connection requests. This table must be ready before `NU_Accept` begins to block waiting for connection attempts.

Parameters

Parameter	Description
socketd	(INT) Specifies the server's socket descriptor.
peer	(struct addr_struct *) On return, this is a pointer to the protocol-specific address of the client.
addrlen	(INT16 *) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, a new socket descriptor is returned from the `NU_Accept` routine with a value greater than or equal to 0. The new socket descriptor contains information specific to the server as well as the client. The original socket descriptor is maintained in its original form in the event that another connection is established later between the same server and a different client. If a connection is not successfully established, a Nucleus status code is returned.



Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NO_TASK_MATCH	The user specified non-blocking and no connection available.
NU_INVALID_PROTOCOL	The combination of family and type parameters for the new socket mapped to a protocol, which were not supported.
NU_NO SOCK_MEMORY	There was not enough memory available to allocate a new socket descriptor structure.
NU_NO_SOCKET_SPACE	There was not space available in the list of socket descriptors for a new descriptor.
NU_NO_PORT_NUMBER	There was not an active port for the socket that was specified which means that a connection never happened.

Example

```

INT socketd;          /* the original socket descriptor */
int new_socket;        /* the new socket descriptor      */
struct addr_struct peer; /* pointer to the client address
                        structure*/

new_socket = NU_Accept(socketd, &peer, 0);

/* if new_socket contains a value greater than or equal to 0,
   the NU_Accept call was successful. new_socket contains the
   socket descriptor, and peer points to the client address
   structure */

```



NU_Add_DNS_Server

```
STATUS NU_Add_DNS_Server (new_dns_server, where);
```

This function will add a new DNS server to the list of DNS servers that will be used when trying to resolve host addresses or host names. The `NU_Get_Host_By_Name`, and `NU_Get_Host_By_Addr` make use of DNS servers to resolve names and addresses.

Parameters

Parameter	Meaning
<code>new_dns_server</code>	(UINT8 *) This is a pointer to the address of a DNS server to be added.
<code>where</code>	(INT) This is a flag that indicates where in the list the DNS server should be added. Valid values for this flag are <code>DNS_ADD_TO_FRONT</code> and <code>DNS_ADD_TO_END</code> . These will cause the DNS server to be added at the head or tail of the list respectively.

Upon successful completion the new DNS server will have been added to the list and `NU_SUCCESS` will be returned. The new DNS server will be available for use by any subsequent calls to `NU_Get_Host_By_Name`, or `NU_Get_Host_By_Addr`. Upon failure one of the following status codes will be returned.

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	Successful completion of the service.
<code>NU_INVALID_PARM</code>	Either the server address is not a valid IP address or the <code>where</code> flag does not contain a valid value.
<code>NU_QUEUE_FULL</code>	The maximum number of DNS servers have already been added.

Example

```
UINT8          dns1[] = {192,200,100,1};
STATUS         status;

/* Add a new DNS server to the head of the list. */
status = NU_Add_DNS_Server(dns1, DNS_ADD_TO_FRONT);

if (status != NU_SUCCESS)
{
    printf("DNS_ERROR: line : %d", __LINE__);
    DEMO_Exit(0);
}
```



NU_Add_Route

```
STATUS NU_Add_Route(ip_dest, mask, gw_addr)
```

This function is used by applications to add a route to the routing table. This function can also be used to add a default route as well as a normal route. Note that a route is added automatically for each MAC layer device as it is initialized, including serial (SLIP and PPP) devices. So it is not necessary to add routes to directly connected hosts.

Parameters

Parameters	Meaning
ip_dest	(UINT8 *) This is the IP address of the destination to which the route is being added. There are three possible cases for this value. It can be a host IP address. It can be a network address. Or it can be an IP address of all 0's. The last case indicates that a default route is being added.
mask	(UINT8 *) This parameter specifies the mask that should be used for this route.
gw_addr	(UINT8 *) This parameter specifies the IP address of the gateway or next hop.

Return Value

Code	Meaning
NU_SUCCESS	The route was successfully added.
NU_INVALID_PARM	One of the parameters is an invalid value.

Example

```
UINT8 subnet[] = {255,255,255,0};
UINT8 netwrk[] = {192,200,100,0};
UINT8 router[] = {208,239,168,195};

NU_Add_Route(netwrk, subnet, router);
```



NU_Bind

```
STATUS NU_Bind (socketd, myaddr, addrlen);
```

This function is responsible for assigning a local address to a socket. It must be called by a server whether a connection-oriented or connectionless transfer is being established. A client, however, only needs to call this function during a connectionless transfer.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
myaddr	(struct addr_struct *) Pointer to the server's protocol-specific address.
addrlen	(INT16) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, NU_Bind updates the local address portion of the socket descriptor and socketd is returned from the NU_Bind routine. If the request was not successful a value less than zero will be returned. The NU_INVALID_SOCKET error condition indicates the socket parameter was not a valid socket value, or it had not been previously allocated via the NU_Socket call.

Example

```
INT socketd;      /* the socket descriptor */

/* the server address structure*/
struct addr_struct *myaddr;

/* the user must fill in the address of the local
machine before making the NU_Bind call */

myaddr->family = NU_FAMILY_IP;    /* Specifies Internet Protocol */
myaddr->port = 4000;               /* local machine's port num */
myaddr->id.is_ip_addr[0] = 192;
myaddr->id.is_ip_addr[1] = 9;      /*local machine's */
myaddr->id.is_ip_addr[2] = 200;    /* 4-digit IP number*/
myaddr->id.is_ip_addr[3] = 3;
myaddr->name = "server_name"

socketd = NU_Bind(socketd, myaddr, 0);
/* if socketd contains a value greater than or equal to 0, the
NU_Bind call was successful and socketd contains the socket
descriptor */
```



NU_Bootp

```
STATUS NU_Bootp (dv_name, bp_ptr);
```

The Bootstrap protocol provides a means for a diskless device to acquire an IP address and configuration information. The Bootstrap Protocol is performed with a single packet exchange between the BOOTP client and the BOOTP Server. The Client sends the BOOTP Request. After receiving the BOOTP Request, the Server sends the BOOTP Reply filled with data that is configured within the BOOTP Server. The BOOTP Client receives the packet and parses out the IP address and any other configuration information provided in the response. Optionally the name of a remote file can be provided in the response.

This file, which might contain configuration data not defined by the Bootstrap Protocol, can then be retrieved. Typically TFTP would be used to retrieve the remote file. The actual processing of the file is left up to the application.

The NU_Bootp service provides the means by which Nucleus NET applications can access the Bootstrap protocol. If required, this function should be called soon after power-up, but after the setting of the device structures as shown in the example at the end of this section. By default support for Bootp will not be included in the Nucleus NET library. The NU_Bootp service is enabled by setting the define INCLUDE_BOOTP in TARGET.H to a NU_TRUE. The default definition found in TARGET.H is shown below.

```
/* By default BOOTP client is not included in the Nucleus Net
   Build. To include it change the 0 below to a 1. See the Nucleus
   Net reference Manual for more information on BOOTP.*/

#define INCLUDE_BOOTP NU_FALSE
```

NU_Bootp can be used for multiple device interfaces to acquire IP addresses. The example below shows how two devices acquire IP addresses by using the NU_Bootp service.

When the NU_Bootp service call is used to acquire an IP address, the IP address that is passed to the NU_Init_Devices service should be set to all 0's.

The only special configuration of the BOOTP server required is that the server be configured such that replies are always sent via broadcast. This is because Nucleus NET can only receive broadcast packets on interfaces that do not yet have an attached IP address. Some BOOTP servers default to sending the reply to the IP address that is being returned. Nucleus NET will reject such a reply.

If the BOOTP server will be returning vendor options in the reply, the user may have to configure the BOOTP client to handle some of the vendor options. This can be accomplished by modifying the function BOOTP_Process_Packets in the file BOOTP.C. Describing the modifications necessary to process vendor options, is beyond the scope of this document.



Parameters

Parameter	Meaning
dv_name	(CHAR *) The name of the device that BOOTP will be used to discover an IP address for.
bp_ptr	(NU_BOOTP_STRUCT*) Pointer to a structure that is used to pass information to and get information from the NU_Bootp service.

The second parameter is of type `NU_BOOTP_STRUCT*`, and is defined below. This structure is used to return configuration information to the application. This structure can be modified to hold vendor options that are returned in a BOOTP reply packet.

```
typedef struct bootp_struct {
    UINT8 bp_ip_addr[4]; /* new IP address of client. */
    UINT8 bp_mac_addr[6]; /* MAC address of client. */
    UINT8 bp_sname[64]; /* optional server host name field. */
    UINT8 bp_file[128]; /* Full filename including path */
    UINT8 bp_siaddr[4]; /* BOOTP server IP address */
    UINT8 bp_giaddr[4]; /* Gateway IP address */
    UINT8 bp_yiaddr[4]; /* Your IP address */
    UINT8 bp_net_mask[4]; /* Net mask for new IP address */
} NU_BOOTP_STRUCT;
```

The only field in the `NU_BOOTP_STRUCT` that might need to be filled in by the application is the `bp_net_mask` field. This will be required if the BOOTP server is not configured to return a mask. If the BOOTP server does return a mask and the application also specifies a mask, then the mask from the server will override the one specified by the application.



Return Value

Code	Meaning
NU_SUCCESS	The operation was successful.
NU_BOOTP_INIT_FAILED	A required resource could not be allocated.
NU_NO_BUFFERS	A buffer to build the bootp request in could not be allocated.
NU_BOOTP_SEND_FAILED	BOOTP could not send the configuration request, possibly driver problem.
NU_BOOTP_RECV_FAILED	BOOTP could not receive the configuration request, NU_Recv_From failed.
NU_BOOTP_ATTACH_IP_FAILED	BOOTP obtained an IP address but was unable to assign it to the network device.
NU_BOOTP_SELECT_FAILED	BOOTP could not receive the configuration reply because NU_Select failed.
NU_BOOTP_FAILED	General BOOTP failure case.
NU_INVALID_PARM	One or more pointer parameters were NU_NULL.

Upon successful completion of this service NU_SUCCESS is returned, and the local host's adapter will have an IP address attached. The BOOTP server may have optionally returned a *filename* to the client. If so, this *filename* is passed up to the application through the NU_BOOTP_STRUCT structure. A file transfer protocol such as TFTP or FTP can then be used to retrieve the file from the server.

Example

```
#define DEVICE_COUNT    2

NU_BOOTP_STRUCT        *bootp_ptr, *bp;
NU_DEVICE              devices[DEVICE_COUNT];

void tcp_server_task(UNSIGNED argc, VOID *argv)
{
    char    c;
    int     socketd, newsock;          /* the socket descriptor */
    struct addr_struct *servaddr; /* server address structure*/
    VOID     *pointer;
    STATUS   status;
    struct addr_struct client_addr;
    UINT32   ipaddr;
    INT16    ret;
    UINT32   ip_addr;
    char     subnet[] = {255,255,255,0};
```



```

/* call network initialization */
if (NU_Init_Net(&System_Memory) != NU_SUCCESS)
{
    #if (OUTPUT_OK)
        printf("error at call to NU_Initialize() from
               tcp_server_task.\n");
    #endif

    NU_Suspend_Task(NU_Current_Task_Pointer());
}

devices[0].dv_name = "NE2000_0";
devices[0].dv_hw.ether.dv_irq = 5;
devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = NE2000_Init;
devices[0].dv_flags = 0;
memcpy (devices[0].dv_ip_addr, "\0\0\0\0", 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);

/* Initialize the ethernet device. */
devices[1].dv_name = "NE2000_1";
devices[1].dv_hw.ether.dv_irq = 11;
devices[1].dv_hw.ether.dv_io_addr = 0x0320L;
devices[1].dv_hw.ether.dv_shared_addr = 0;
devices[1].dv_init = NE2000_Init;
memcpy (devices[1].dv_ip_addr, "\0\0\0\0", 4);
memcpy (devices[1].dv_subnet_mask, subnet, 4);
devices[1].dv_flags = 0;

/* Initialize the hardware interfaces. */
NU_Init_Devices(devices, 2);

/* Allocate Memory for the NU_BOOTP_STRUCT. The memory for the
structure is allocated on based on the number of Devices that
will have the Bootp Service performed on. */

status = NU_Allocate_Memory(&System_Memory, (VOID **)&bootp_ptr,
                           sizeof(NU_BOOTP_STRUCT)
                           * DEVICE_COUNT, NU_NO_SUSPEND);

if ( status != NU_SUCCESS )
{
    printf("Can not allocate memory for NU_BOOTP_STRUCT.\n");
    DOS_Exit(-1);
}

```



```

/* set all BOOTP fields to zero value */
memset(bootp_ptr, 0, sizeof(NU_BOOTP_STRUCT)* DEVICE_COUNT);

/* put in user callback function for the vendor options. */
/* bootp_ptr struct above is left blank unless the caller
   wanted to pass requests to the BOOTP server. */

ret = NU_Bootp(devices[0].dv_name, bootp_ptr);

if( ret != NU_SUCCESS )
{
    printf("Bootp failed to resolve an IP address.\n");
    DOS_Exit(-1);
}

/* Increment the NU_BOOTP_STRUCT to the next device. */
bootp_ptr++;

ret = NU_Bootp(devices[1].dv_name, bootp_ptr);

if( ret != NU_SUCCESS )
{
    printf("Bootp failed to resolve an IP address.\n");
    DOS_Exit(-1);
}
}

```



NU_Close_Socket

```
STATUS NU_Close_Socket (socketd);
```

This function is responsible for breaking the given socket connection on TCP or a socket allocation from UDP or Raw IP communication. It must be called by both the client and the server to terminate a connection whether a connection-oriented or connectionless transfer had been established. When using a TCP connection, the `NU_Close` service call will suspend until the connection is completely closed. In the case of UDP services, the socket and port entries are closed immediately.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the <code>NU_Socket</code> call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.
NU_SUCCESS	Successful close.

Example

```
INT socketd;      /* the socket descriptor */
int status;       /* return status of the NU_Close call */

status = NU_Close_Socket(socketd);
/* if status equals NU_SUCCESS, the NU_Close_Socket
   call was successful and the connection is terminated */
```



NU_Connect

```
STATUS NU_Connect (socketd, servaddr, addrlen);
```

This function is responsible for establishing a connection request from a client. Only the client must call the `NU_Connect()` routine. When a connection-oriented transfer is being established, the client and server may exchange messages agreeing on such items as buffer size and amount of data to be transferred. Connection oriented transfers are only performed via the TCP interface; so, the only valid data transfer services that can be used with `NU_Connect` are `NU_Send` and `NU_Recv`.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
servaddr	(struct addr_struct*) Pointer to the server's protocol-specific address.
addrlen	(INT16) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, a socket descriptor is returned from the `NU_Connect` routine with a value greater than or equal to 0. If the connect call is not successful, a Nucleus status code is returned.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the <code>NU_Socket</code> call.
NU_NO_PORT_NUMBER	Was not able to make a Port. Look at <code>Connect_sock</code> .
NU_NO_HOST_NAME	The name of the host was not specified (used for look-up in the machine table set up from the <code>defmachinfo</code> structure) and no other address (IP and port) values were specified.
NU_INVALID_PARM	The server address parameter was not valid.
NU_NOT_CONNECTED	The connection attempt failed.
NU_NO SOCK_MEMORY	Memory was not available to allocate the necessary structures for making a connection.



Example

```
int status; /* return status of the NU_Connect function */
INT socketd; /* the socket descriptor */
struct addr_struct *servaddr; /* the server address structure */
.
.
.
/* the user must fill in the address of the server with which it
   is requesting a connection before making the NU_Connect call */

servaddr->family = NU_FAMILY_IP; /* Internet Protocol */
servaddr->port = 23; /* Telnet's port number */
servaddr->id.is_ip_addr[0] = 192;
servaddr->id.is_ip_addr[1] = 9; /* local machine's */
servaddr->id.is_ip_addr[2] = 200; /* 4-digit IP number */
servaddr->id.is_ip_addr[3] = 3;
servaddr->name = "server_name"

status = NU_Connect(socketd, servaddr, 0);

/* if status is greater than or equal to 0, the NU_Connect call
   was successful and the client is connected to the specified
   server */
```



NU_Delete_DNS_Server

```
STATUS NU_Delete_DNS_Server (dns_ip);
```

This function will delete DNS server from the list of DNS servers that will be used when trying to resolve host addresses or host names. See Chapter 5 for more information on DNS.

Parameters

Parameter	Meaning
dns_ip	(UINT8 *) This is a pointer to the address of the DNS server to be deleted.

Upon successful completion the DNS server will have been deleted from the list and NU_SUCCESS will be returned. Upon failure one of the following status codes will be returned.

Return Value

Code	Meaning
NU_SUCCESS	Successful completion of the service.
NU_INVALID_PARM	The IP address was not valid. It was not found in the list of DNS servers.

Example

```
UINT8          dns1[] = {192,200,100,1};
STATUS         status;

status = NU_Delete_DNS_Server(dns1);
if (status != NU_SUCCESS)
{
    printf("DNS_ERROR: line : %d", __LINE__);
}
```



NU_Dhcp

```
STATUS NU_Dhcp(ds_ptr, dv_name);
```

This service is used to dynamically configure the IP address of an embedded device. It can also be used to retrieve other information from a DHCP server, such as a mask, default gateway, etc. Please see Chapter 5 for an extended discussion on the *Dynamic Host Configuration Protocol*.

Parameters

Parameter	Meaning
ds_ptr	(DHCP_STRUCT*) This service is used to dynamically configure the IP address of an embedded device. It can also be used to retrieve other information from a DHCP server, such as a mask, default gateway, etc.
dv_name	(CHAR*) This parameter is the name of the device for which an IP address is needed.

Upon success completion, NU_SUCCESS is returned, the local host's IP address is initialized and the NU_DHCP_STRUCT as shown above is filled. The NU_DHCP_STRUCT can be used in the application to access all DHCP information. Otherwise, a status code is returned.

Return Value

Code	Meaning
NU_SUCCESS	The local host's IP address is initialized and the NU_DHCP_STRUCT is filled.
NU_NO SOCK_MEMORY	There was not enough memory available to allocate a new socket descriptor structure.
NU_INVALID_PROTOCOL	The combination of family and type parameters for the new socket mapped to a protocol, which was not supported.
NU_NO_SOCKET_SPACE	All sockets are in use. The maximum number of sockets can be increased by changing the definition of NPORTS and NUPOINTS in TARGET.H.
NU_NO_BUFFERS	No buffers available for allocation.
NU_DHCP_INIT_FAILED	A resource (memory, socket, etc.) was not available or could not be initialized properly.
NU_DHCP_REQUEST_FAILED	A DHCP request was sent but no response was received.



Example

```

#define DHCP_DEV_COUNT    2
NU_DHCP_STRUCT    dhcp_ptr, *dsp;
NU_DEVICE    devices[DEVICE_COUNT];

/* These are the dhcp options desired. Each option is specified by
   three items, an option id, an option length, and an option
   value. The following specifies two options. The first option is
   to request that the DHCP server return some specific parameters.
   There are three such parameters (the length). The three
   requested are DHCP_MASK, DHCP_ROUTE, and DHCP_DNS. The second
   option, DHCP_HOSTNAME, is to specify a name that will be sent to
   the server. The name has 11 characters including the null
   terminator. The value is "DHCPCLIENT" + null terminator. */

UINT8 dhcp_options[] = {DHCP_REQUEST_LIST,3,DHCP_NETMASK,
                        DHCP_ROUTE, DHCP_DNS, DHCP_HOSTNAME, 11,
                        'D','H','C','P','C','L','I','E','N','T',
                        0};

devices[0].dv_name = "NE2000_0";
devices[0].dv_hw.ether.dv_irq = 5;
devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = NE2000_Init;
devices[0].dv_flags = 0;
memcpy(devices[0].dv_ip_addr, "\0\0\0\0", 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);

devices[1].dv_name = "NE2000_1";
devices[1].dv_hw.ether.dv_irq = 11;
devices[1].dv_hw.ether.dv_io_addr = 0x0320L;
devices[1].dv_hw.ether.dv_shared_addr = 0;
devices[1].dv_init = NE2000_Init;
devices[1].dv_flags = 0;
memcpy(devices[1].dv_ip_addr, "\0\0\0\0", 4);
memcpy (devices[1].dv_subnet_mask, subnet, 4);

/* Initialize the devices. */
NU_Init_Devices(devices, DHCP_DEV_COUNT);

/* allocate memory for the DHCP structures. */
status = NU_Allocate_Memory(&System_Memory, (VOID **)&dhcp_ptr,
                           sizeof(NU_DHCP_STRUCT)
                           * DHCP_DEV_COUNT, NU_NO_SUSPEND);

/* set all DHCP fields to zero value */
memset(dhcp_ptr, 0, sizeof(NU_DHCP_STRUCT) * DHCP_DEV_COUNT );

```



```
/* put in user callback function for the vendor options. */
for( i = 0, dsp = dhcp_ptr; i < DEVICE_COUNT; i++, dsp++ )
{
    /* Specify the DHCP options desired. */
    dsp->dhcp_opts = dhcp_options;
    dsp->dhcp_opts_len = sizeof(dhcp_options);
}

/* dhcp_ptr struct above is left blank unless the caller wanted
   to pass requests to the DHCP server. Get the IP address and
   information from the DHCP Server for the first device.  */
ret = NU_Dhcp(dhcp_ptr, devices[0].dv_name);

/* If NU_Dhcp returns NU_SUCCESS then the IP address has been
   initialized and DHCP structure filled */
```



NU_Dhcp_Release

```
STATUS NU_Dhcp_Release( ds_ptr, dv_name);
```

This function is responsible for releasing an IP address obtained from a Dynamic Host Configuration Protocol Server. The parameters contain information that was originally provided by the Dynamic Host Configuration Protocol Server.

Parameters

Parameter	Meaning
ds_ptr	(DHCP_STRUCT *) This parameter is a pointer the DHCP structure that will be filled when the DHCP reply packet is processed.
dv_name	(CHAR *) This parameter is the name of the device for which an IP address is to be released.

Upon successful completion of the NU_Dhcp_Release service, the value of NU_SUCCESS shall be returned.

Example

```
#define DHCP_DEV_COUNT          2
NU_DHCP_STRUCT                 *dhcp_ptr, *dsp;
NU_DEVICE                       devices[DEVICE_COUNT];

devices[0].dv_name = "NE2000_0";
devices[0].dv_hw.ether.dv_irq = 5;
devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = NE2000_Init;
devices[0].dv_flags = 0;
memcpy(devices[0].dv_ip_addr, "\0\0\0\0", 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);
```



```

devices[1].dv_name = "NE2000_1";
devices[1].dv_hw.ether.dv_irq = 11;
devices[1].dv_hw.ether.dv_io_addr = 0x0320L;
devices[1].dv_hw.ether.dv_shared_addr = 0;
devices[1].dv_init = NE2000_Init;
devices[1].dv_flags = 0;
memcpy(devices[1].dv_ip_addr, "\0\0\0\0", 4);
memcpy(devices[1].dv_subnet_mask, subnet, 4);

/* Initialize the devices. */
NU_Init_Devices(devices, DHCP_DEV_COUNT);

/* allocate memory for the DHCP structures. */
status = NU_Allocate_Memory(&System_Memory, (VOID **)&dhcp_ptr,

/* set all DHCP fields to zero value */
memset(dhcp_ptr, 0, sizeof(NU_DHCP_STRUCT) * DHCP_DEV_COUNT );

/* put in user callback function for the vendor options. */
for( i = 0, dsp = dhcp_ptr; i < DEVICE_COUNT; i++, dsp++ )
{
    /* Specify the DHCP options desired. */
    dsp->dhcp_opts = dhcp_options;
    dsp->dhcp_opts_len = sizeof(dhcp_options);
}

/* dhcp_ptr struct above is left blank unless the caller wanted to
   pass requests to the DHCP server. Get the IP address and
   information from the DHCP for them first device. */
ret = NU_Dhcp(dhcp_ptr, devices[0].dv_name);

/* If NU_Dhcp returns NU_SUCCESS then the IP address has been
   initialized and the DHCP structure filled */

/* Release the device 0 DHCP obtained device. */
status = NU_Dhcp_Release(dhcp_ptr, devices[0].dv_name);

/* If NU_Dhcp_Release returns NU_SUCCESS then the IP address has
   been successfully released */

```



NU_Fcntl

```
STATUS NU_Fcntl (socketd, command, argument);
```

This function is responsible for setting the block flag associated with the socket descriptor. It may be called by both the client and the server to wait for data during a NU_Recv call.

Parameters

Parameters	Meaning
socketd	(INT) Specifies a socket descriptor.
command	(INT16) Specifies the command requested by the application. Currently the only valid command is NU_SETFLAG.
argument	(INT16) Specifies the command option. The only valid arguments are NU_BLOCK to enable blocking and NU_FALSE to disable blocking.

Return Value

Code	Meaning
NU_SUCCESS	Successful action performed on the block flag.
NU_NO_ACTION	No action was processed by this function.
NU_INVALID_SOCKET	The socket parameter was not a valid socket value.

Example

```
INT socketd; /* the socket descriptor */
int status; /* return status of the NU_fcntl call */
.
.
status = NU_Fcntl(socketd, NU_SETFLAG, NU_BLOCK);

/* if status equals NU_SUCCESS, the NU_Fcntl call was
   successful and the block flag is enabled */
```



NU_FD_Check

```
INT NU_FD_Check(socket, fd);
```

This function is responsible for checking the specified bit in a bitmap to see if it is set. It is generally used in conjunction with `NU_Select` to check the bitmap returned. See `NU_Select` for more information.

Parameters

Parameter	Meaning
socket	(INT) Specifies the index of the bit to check.
fd	(FD_SET *) Pointer to the bitmap which will be checked.

Return Value

A value of `NU_TRUE` is returned when the specified bit is set, otherwise a value of `NU_FALSE` is returned.

Example

```
int    i;
.
.
.
/* Make sure all bits are initially set to 0 */
NU_FD_Init(&readfs);

/* Set the bit for the first three sockets. */
for (i=0; i<3; i++)
{
    NU_FD_Set(i, &readfs);
}

/* Now call NU_Select to see which, if any, have data. */
if (NU_Select(max_sockets, &readfs, &writefs,
    &exceptfs, NU_NO_SUSPEND) == NU_SUCCESS)
{
    /* At least one of the sockets contained received
       data. Check all of them to find those that have data. */
    for (i=0; i<3; i++)
    {
        if (NU_FD_Check(i, &readfs)==NU_FALSE)
            continue;
        .
        .
        .
    }
}
```



NU_FD_Init

```
void NU_FD_Init(fd);
```

This function is responsible for initializing the bitmaps that are used with the `NU_Select` function. This call will initialize each bit in the bitmap to 0.

Parameters

Parameter	Meaning
fd	(FD_SET *) This parameter is a pointer to the bitmap that will be initialized.

Return Value

None.

Example

In this example, our node is acting as a server.

```
int    i;
.
.
.
/* Make sure all bits are initially set to 0 */
NU_FD_Init(&readfs);

/* Set the bit for the first three sockets. */
for (i=0; i<3; i++)
{
    NU_FD_Set(i, &readfs);
}

/* Now call NU_Select to see which, if any, have data. */
if (NU_Select(max_sockets, &readfs, &writefs,
    &exceptfs, NU_NO_SUSPEND) == NU_SUCCESS)
{
    /* At least one of the sockets contained received
    data. Check all of them to find those that have data. */
    for (i=0; i<3; i++)
    {
        if (NU_FD_Check(i, &readfs)==NU_FALSE)
            continue;
    }
}
```



NU_FD_Reset

```
void NU_FD_Reset (socket, fd);
```

This function is responsible for resetting one bit in the specified bitmap to 0. This service is generally used in conjunction with `NU_Select`. Please see `NU_Select` for more information.

Parameters

Parameter	Meaning
socket	(INT) Specifies the index the bit to be reset.
fd	(FD_SET *) Pointer to the bitmap in which the bit will be reset.

Return Value

No return value is required. Its usage is the same as `NU_FD_Set()`.



NU_FD_Set

```
void NU_FD_Set(socket, fd);
```

This function is responsible for setting the specified bit of a bitmap to 1. This service is generally used in conjunction with `NU_Select`. Please see `NU_Select` for more information.

Parameters

Parameter	Meaning
socket	(INT) Specifies the index of the bit to be set.
fd	(FD_SET *) Pointer to the bitmap in which the bit will be set.

Return Value

No return value is required.

Example

```
int    i;
.
.
.
/* Make sure all bits are initially set to 0 */
NU_FD_Init(&readfs);

/* Set the bit for the first three sockets. */
for (i=0; i<3; i++)
{
    NU_FD_Set(i, &readfs);
}

/* Now call NU_Select to see which, if any, have data. */
if (NU_Select(max_sockets, &readfs, &writefs,
    &exceptfs, NU_NO_SUSPEND) == NU_SUCCESS)
{
    /* At least one of the sockets contained received data. Check
       all of them to find those that have data. */
    for (i=0; i<3; i++)
    {
        if (NU_FD_Check(i, &readfs)==NU_FALSE)
            continue;
    }
}
```



NU_Get_DNS_Servers

```
INT NU_Get_DNS_Servers(dest, size);
```

This function will return the list of DNS servers that are being used by Nucleus NET.

Parameters

Parameter	Meaning
dest	(UINT8 *) This is a pointer to a memory block where the list of DNS server IP addresses will be copied.
size	(INT) This is the size of the block of memory pointed to by dest.

Upon successful completion the list of DNS servers will have been copied into the block of memory pointed to by dest. If dest is not large enough to hold the entire list, only as many addresses as will fit will be copied into dest. The number of addresses copied to dest will be returned upon success. Upon failure one of the following status codes will be returned.

Return Value

Code	Meaning
NU_INVALID_PARM	Either dest is a NULL pointer or size is less than 4.

Example

```
UINT8      dns1[] = {192,200,100,1};
STATUS     status;
UINT8      dns_list[5 * 4]; /* 5 ip addresses. */

/* Add a new DNS server to the head of the list. */
status = NU_Add_DNS_Server(dns1, DNS_ADD_TO_FRONT);

if (status != NU_SUCCESS)
{
    printf("DNS_ERROR: line : %d", __LINE__);
    DEMO_Exit(0);
}

/* Now make sure the IP address for the server was added. */
status = NU_Get_DNS_Servers(dns_list, sizeof(dns_list));
if (status <= 0)
{
    printf("DNS_ERROR: line : %d", __LINE__);
    DEMO_Exit(0);
}
```



NU_Get_Host_By_Addr

```
STATUS NU_Get_Host_By_Addr(addr, len, type, host_entry);
```

This function returns a pointer to a host structure based on the IP address specified. First the local “hosts file” is searched for the specified host. If not found in the local cache the Domain Name System (DNS) is used to resolve the host via the network. Please see Chapter 5, Domain Name System (DNS), for more information on DNS and how to configure DNS for your network.

Parameters

Parameter	Meaning
addr	(CHAR *) This parameter specifies the IP address of the host to be searched for.
len	(INT) This is the length of addr, the first parameter. A value of 4 should always be used.
type	(INT) This parameter specifies the type of address in parameter one. It should always be NU_FAMILY_IP.
host	(NU_HOSTENT *) This parameter is a pointer to a host structure. The contents of host are undefined upon function entry. Upon successful completion, this structure will be filled in with the host information.

Return Value

Code	Meaning
NU_SUCCESS	Indicates successful completion of the service (the host was found).
NU_INVALID_PARM	One of the parameters is a NULL pointer.
NU_NOT_A_HOST	The host could not be located.

On a successful completion, the structure pointed to by host has been filled in with the host information, name, IP address, etc.

Example

```
char          ip_addr[] = {206,202,34,80};
NU_HOSTENT    hentry;

if (NU_Get_Host_By_Addr(ip_addr, 4, NU_FAMILY_IP, &hentry)
    != NU_SUCCESS)
    exit(0);
```



NU_Get_Host_By_Name

```
STATUS NU_Get_Host_By_Name(name, host_entry);
```

This function returns a pointer to a host structure based on the name specified. First the local “hosts file” is searched for the specified host. If not found in the local cache, the Domain Name System (DNS) is used to resolve the host via the network. Please see Chapter 5, Domain Name System (DNS), for more information on DNS and how to configure DNS for your network.

Parameters

Parameter	Meaning
name	(CHAR *) This parameter specifies the name of the host to be searched for.
host_entry	(NU_HOSTENT *) This parameter is a pointer to a host structure. The contents of <code>host</code> are undefined upon function entry. Upon successful completion, this structure will be filled in with the host information.

Return Value

Code	Meaning
NU_SUCCESS	Indicates successful completion of the service (the host was found).
NU_INVALID_PARM	One of the parameters is a NULL pointer.
NU_NOT_A_HOST	The host could not be located.

On a successful completion the structure pointed to by `host` has been filled in with the host information, name, IP address, etc.



NU_Get_Peer_Name

```
STATUS NU_Get_Peer_Name (socketd, peer, addr_length);
```

his function is responsible for returning the remote endpoint of a specified connection. This function may be called by a server to determine the address of the client to which it is connected.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
peer	(struct sockaddr_struct *) A pointer to a structure containing the remote client's address.
addr_length	(INT16 *) Return pointer for the length of the peer structure returned.

Return Value

Code	Meaning
NU_SUCCESS	Successful retrieval of a socket option.
NU_BAD_SOCKETD	The socket descriptor was invalid.
NU_NOT_CONNECTED	The specified socket descriptor did not have a client connected to it.
NU_INVALID_PARM	The address length is invalid.

Example

```
int  status;      /* status of Get Peer Name call */
INT  socketd;     /* the socket descriptor */
struct sockaddr_struct peer; /* holds the client remote address */
int  addr_length; /* length of peer structure */
.
.
.
status = NU_Get_Peer_Name(socketd, &peer, &addr_length);

/* if status is equal to NU_SUCCESS, then peer contains the
   remote address of the client to which the server is connected
   for the specified socket descriptor and addr_length contains
   the length of the peer structure */
```



NU_Getsockopt

```
STATUS NU_Getsockopt(socketd, level, optname, optval, optlen);
```

This function is responsible for returning the status of an option for a specified socket. Currently the only supported options are for toggling broadcasting and multi-casting support on a socket. Other options will be supported in the future. The `NU_Setsockopt` service can be used to set socket options.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
level	(INT) Specifies the protocol level. The only valid entries for this parameter are <code>SOL_SOCKET</code> and <code>IPPROTO_IP</code> .
optname	(INT) Specifies an option. Currently the only valid values for this parameter are <code>SO_BROADCAST</code> , and <code>IP_MULTICAST_TTL</code> .
optval	(void *) Pointer to the location where the option status can be written.
optlen	(INT*) <code>optlen</code> should contain the size of the location pointed to by <code>optval</code> . Upon returning specifies the size in bytes of the data copied into the location pointed to by <code>optval</code> .

Upon successful completion of this service, `optval` will contain the status of the option specified by `optname`, and `optlen` will contain the size of the data was placed into the location pointed to by `optval`. This function also returns a status code.

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	Successful retrieval of a socket option.
<code>NU_INVALID_SOCKET</code>	The specified socket descriptor is invalid.
<code>NU_NOT_CONNECTED</code>	A connection does not exist on the specified socket.
<code>NU_INVALID_LEVEL</code>	The value specified in the <code>level</code> parameter is invalid.
<code>NU_INVALID</code>	<code>optval</code> is a null pointer.
<code>NU_INVALID_OPTION</code>	The value specified in the <code>optname</code> parameter is invalid.



The following is a list of levels and the associated options that are currently supported by `NU_Getsockopt`.

SOL_SOCKET Level	Option
<code>SO_BROADCAST</code>	Check the broadcast status of a socket. When a socket is created the ability to send broadcasts is enabled by default.

IPPROTO_IP Level	Options
<code>IP_MULTICAST_TTL</code>	Check the TTL (Time To Live) value that is used for multicast packets that are sent using this socket. The default TTL is 1. This keeps routers from forwarding the multicast datagrams beyond the local network.

Example

```

INT socketd; /* the socket descriptor */
STATUS status;
INT optstatus = 0;
INT optlen;
.
.
.
optlen = sizeof(optlen);

/* Retrieve the status of the BROADCAST socket option. */
status = NU_Getsockopt(socketd, SOL_SOCKET, SO_BROADCAST,
                      &optstatus, &optlen);

/* If the call was successful and broadcasting is disabled then
enable broadcasting. */
if (status == NU_SUCCESS && !(optstatus & SO_BROADCAST))
{
    optstatus = SO_BROADCAST;
    NU_Setsockopt(socketd, SOL_SOCKET, SO_BROADCAST, &optstatus,
                  sizeof(INT16));
}

/* Send the datagram. */
bytes_sent = NU_Send_To(socketd, buffer, strlen(buffer),
                        0, broadaddr, servlen);

```



NU_Ioctl

```
STATUS NU_Ioctl(optname, option, optlen);
```

This function is responsible for performing specialized functions on an interface or other object. Currently the only supported options are for retrieving the IP address associated with a physical layer device, and retrieving the IP address of the foreign host at the other end of a Point to Point connection.

Parameters

Parameter	Meaning
optname	(INT) Specifies an option. Currently, the only valid values for this parameter is <code>IOCTL_GETIFADDR</code> and <code>IOCTL_GETIFDSTADDR</code> .
option	(<code>SCK_IOCTL_OPTION *</code>) Return pointer for option status.
optlen	(INT) Specifies the size in bytes of the location pointed to by option.

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	Indicates successful completion of the service.
<code>NU_INVALID</code>	The value specified in the <code>optlen</code> parameter was not large enough to store the option status.
<code>NU_INVALID_PARM</code>	The device specified by the option parameter is invalid.
<code>NU_INVALID_OPTION</code>	The value specified in the <code>optname</code> parameter is invalid.



Example

```

int socketd;          /* the original socket descriptor */
DEV_DEVICE devices[1];
NU_IOCTL_OPTION option;
IP_MREQ mgroup;
.
.
.
/* This example shows how to find the IP address attached
   to a particular device. */

/* Point to the name of the device that we wish to know
   the IP address of. */
option.s_optval = devices[0].dv_name;

/* Call NU_Ioctl to get the IP address. */
if (NU_Ioctl(IOCTL_GETIFADDR, &option, sizeof(option))
    != NU_SUCCESS)
    DEMO_Exit(0);

/* Copy the retrieved IP address. */
memcpy (&mgroup.sck_inaddr, option.s_ret.s_ipaddr, 4);

```



NU_Init_Devices

```
STATUS NU_Init_Devices(devices, dev_count);
```

This function is responsible for initializing all of the devices that will be used by Nucleus NET. It will be hard coded to set up the devices based on the developer's requirements.

Parameters

Parameters	Meaning
devices	(NU_DEVICE *) Array of device information structures.
dev_count	(INT16) Number of device structures in device array.

Return Value

If the device initialization was successful, NU_SUCCESS is returned.

Example

```
NU_DEVICE devices[1];
devices[0].dv_name = NU_Argv[1];
devices[0].dv_hw.ether.dv_irq = 0;      /* Not used */
devices[0].dv_hw.ether.dv_io_addr = 0; /* Not used */
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = VDRV_Init;
memcpy (devices[0].dv_ip_addr, ip_addr, 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);
devices[0].dv_flags = 0;
.
.
.
status = NU_Init_Devices(devices, 1);
/* If status is NU_SUCCESS, then the devices specified were
   successfully initialized */
```



NU_Init_Net

```
STATUS NU_Init_Net(mem_pool);
```

This is the first function that is called before utilizing Nucleus NET. It is responsible for setting up the internal data structures for the network stack and allocating any resources required by Nucleus NET.

Parameters

Parameter	Meaning
mem_pool	(NU_MEMORY_POOL *) This parameter specifies a Nucleus PLUS memory pool from which memory for the Nucleus NET buffers and memory for structures shared by both hardware devices and the stack will be allocated.

The single parameter to `NU_Init_Net` can be used to specify a memory pool of non-cached memory. This is useful because Nucleus NET drivers share the same pool of memory buffers as does the stack. On some architectures the device will be given a pointer to a memory buffer and will write received data and status information directly to the buffer. On such architectures it is necessary to frequently flush the cache before accessing buffers that have just been returned by the driver to the stack. As an alternative a non-cached memory pool can be specified from which to allocate Nucleus NET buffers and data structures that can be accessed by both hardware and the driver. If such a non-cached memory pool is not needed or desired, then any memory pool will do, including the `System_Memory` pool.

Return Value

Code	Meaning
NU_SUCCESS	Indicates the socket option was set successfully.
NU_MEM_ALLOC	The <code>System_Memory</code> pool has been exhausted.
NU_INVALID	A general-purpose error condition. This generally indicates that a required resource (task, semaphore, etc.) could not be created.
NU_DHCP_INIT_FAILED	Failed to initialize the DHCP module.



Example

```
int status; /* the status that will be returned */
NU_MEMORY_POOL System_Memory;

/* Create a system memory pool that will be used to allocate
   task stacks, queue areas, etc. */
status = NU_Create_Memory_Pool(&System_Memory, "SYSMEM",
                               first_available_memory, 400000,
                               50, NU_FIFO);

if (status != NU_SUCCESS)
{
    printf ("Can not create the System Memory Pool.\n");
    exit (0);
}

status = NU_Init_Net(&System_Memory);

/* Was the stack successfully initialized? */
if (status != NU_SUCCESS)
    exit(0);
```



NU_Is_Connected

```
STATUS NU_Is_Connected(socketd);
```

This function is responsible for determining if a connection has been established on the input socket descriptor. This service can only be used with sockets that are using TCP.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.

Return Value

Code	Meaning
NU_TRUE	The socket is connected.
NU_FALSE	The socket is not connected.
NU_INVALID_SOCKET	The value specified in the <code>socketd</code> parameter is invalid.

Example

```
INT socketd;      /* A socket descriptor */
.
.
.
status = NU_Is_Connected(socketd);
```



NU_Listen

```
STATUS NU_Listen (socketd, backlog);
```

This function is responsible for indicating that the server is willing to accept connection requests from clients. This function only needs to be called by the server when a connection-oriented transfer is being established.

Parameters

Parameters	Meaning
socketd	(INT) Specifies the server's socket descriptor.
backlog	(UINT16) Specifies the number of requests which can be queued for this server.

Return Value

Code	Meaning
NU_SUCCESS	Successful listen.
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NOT_A_TASK	No task was running. NU_Listen was probably called from an interrupt thread that did not perform the proper context save operation.
NU_NO SOCK_MEMORY	There was not enough memory for Nucleus to allocate the necessary structures to be ready for a connection.

Example

```
INT socketd;    /* the socket descriptor */
.
.
.
status = NU_Listen(socketd, 10);
/* if status equals NU_SUCCESS, the NU_Listen call
   was successful and the server is prepared to
   accept connection requests */
```



NU_Ping

```
STATUS NU_Ping (dest_ip, timeout);
```

This function will ping an IP address, send an ICMP echo request, once and wait for a response. A timeout is used for returning from this function if the foreign host does not reply to the ping. This value can be supplied or the default timeout of 5 seconds can be used. This service is useful for determining if a host on a network is up and running prior to starting some type of network communication with them. Specifically in the case of UDP were it is unknown by the protocol if the host is available.

Parameters

Parameter	Meaning
dest_ip	(UINT8 *) This is a pointer to the address of the host to ping.
timeout	(UINT32) The timeout value is supplied in Nucleus PLUS clock ticks. If a value of zero is supplied a default timeout of 5 seconds will be used.

Return value

Code	Meaning
NU_SUCCESS	The foreign host replied to the ping request.
NU_TIMEOUT	A ping reply was not received within the timeout period.
NU_NO_ACTION	NET was unable to send the ping request. Possibly due to no route available to the foreign host.
NU_NO_BUFFERS	The ping request could not be sent because of a lack of internal NET buffers.
NU_MEM_ALLOC	The ping request was sent because NET could not allocate the needed memory to complete the service.



NU_Rarp

```
STATUS NU_Rarp(device_name);
```

This function is responsible for dynamically resolving the IP addresses for diskless workstations when only their physical hardware address is known. To make use of this service there must be a RARP server on the local network, who can resolve the address.

Parameters

Parameter	Meaning
device_name	(CHAR *) Specifies the name of the device for which the IP address should be resolved.

Return Value

Code	Meaning
NU_SUCCESS	Indicates successful completion of the service.
NU_INVALID_PARM	A device with device_name could not be found.
NU_HOST_UNREACHABLE	The device with device_name has not been initialized. The NU_Init_Devices service should be used before calling NU_Rarp.
NU_NO_BUFFERS	There were no buffers available for building the RARP request packet.
NU_MEM_ALLOC	Indicates a memory allocation failure.
NU_RARP_INIT_FAILED	RARP failed to resolve the IP address. Possible problems are that there are no RARP servers on the local network or the RARP server does not know the device's IP address.



Example

```

/* An IP address of all 0's should be used when the IP address
   is to be resolved by RARP. */
char      ip_addr[] = {0,0,0,0};
char      subnet[] = {255,255,255,0};
NU_DEVICE devices[1];

/* Initialize the network stack. */
if(NU_Init_Net(&System_Memory))
{
    printf("error at call to NU_Initialize() from
           TCP_client_task");
    DOS_Exit(0);
}

/* Register the MAC layer device with the stack. */
devices[0].dv_name = "NE2000_0";
devices[0].dv_hw.ether.dv_irq = 5;
devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = NE2000_Init;
devices[0].dv_flags = 0;
memcpy (devices[0].dv_ip_addr, ip_addr, 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);

NU_Init_Devices(devices, 1);

/* Now that the stack and the hardware device has been
   initialized NU_Rarp can be called to discover the IP
   address for the device. */
if (NU_Rarp(devices[0].dv_name) != NU_SUCCESS)
{
    printf("RARP failed.\n");
    NU_Suspend_Task(NU_Current_Task_Pointer());
}

```



NU_Recv

```
STATUS NU_Recv (socketd, buff, nbytes, flags);
```

This function is responsible for receiving data across a network during a connection oriented transfer. It may be called by both the client and the server.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(char *) Pointer to the data buffer.
nbytes	(UINT16) Specifies the number of bytes of data.
flags	(INT16) N/A

On a successful completion, the `NU_Recv` routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned. The status codes returned and their associated meanings are defined below.

Return Value

Code	Meaning
NU_NOT_CONNECTED	The connection is broken for some reason. Stop using the socket; it is best to close it.
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the <code>NU_Socket</code> call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.
NU_NOT_A_TASK	No task was running. <code>NU_ODH_Recv</code> was probably called from an interrupt thread that did not perform the proper context save operation.

Example

```
INT socketd;          /* the original socket descriptor */
char *buff;           /* pointer to the data buffer */
int nbytes = 80;      /* number of bytes to be sent */
int count;            /* number of bytes successfully transferred */
.
count = NU_Recv(socketd, buff, nbytes, 0);
/* if count contains a value greater than or equal to 0, then
   count is the number of bytes received and the data is at
   the location pointed to by buff */
```



NU_Recv_From

```
STATUS NU_Recv_From (socketd, buff, nbytes, flags, from, addrlen);
```

This function is responsible for receiving data across a network during a connectionless transfer. It may be called by both the client and the server.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(CHAR *) Pointer to the data buffer. NOTE: because the nbytes parameter is not currently used, the buffer must be large enough to hold the largest datagram that the host expects to receive. In the case of ethernet, this can never be larger than 1500 bytes.
nbytes	(INT16) Specifies the number of bytes of data. NOTE: this parameter is not currently used. Instead the last datagram received is copied as a whole into the location pointed to by the buff parameter.
flags	(INT16) N/A
from	(struct addr_struct *) Pointer to the source protocol-specific address structure.
addrlen	(INT16 *) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, the NU_Recv_From routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.
NU_NO_DATA_TRANSFER	The data transfer was not completed.



Example

```

INT socketd;      /* the original socket descriptor */
char *buff;       /* pointer to the data buffer */
struct addr_struct myaddr; /* address of server */
struct addr_struct cliaddr; /* address of client */
int clilen; /* length of the client address (unused) */
int count; /* number of bytes successfully transferred */
.
.
/* Build a socket. */
.
.
/* Allocate memory for my address structure. */
.
.
/* Build the my address structure. */
myaddr->family = NU_FAMILY_IP; /* Internet Protocol */
myaddr->port = 7000; /* my server port number */

myaddr->id.is_ip_addr[0] = 192;
myaddr->id.is_ip_addr[1] = 9; /* my machine's */
myaddr->id.is_ip_addr[2] = 200; /* 4-digit IP number */
myaddr->id.is_ip_addr[3] = 1;
myaddr->name = "my_name"

/* Allocate memory for client address structure. */
.
.
/* Build the my address structure. */

/* Internet Protocol */
cliaddr->family = NU_FAMILY_IP;

/* client's port number, will be filled in by socket library */
cliaddr->port = 0;

/* client machine's */ /* 4-digit IP number */
cliaddr->id.is_ip_addr[0] = 192;
cliaddr->id.is_ip_addr[1] = 9;
cliaddr->id.is_ip_addr[2] = 200;
cliaddr->id.is_ip_addr[3] = 4;
cliaddr->name = "client_name"

/* Bind our address to the socket. */
NU_Bind(socketd, myaddr, 0);

/* Wait for data. */
count = NU_Recv_From (socketd, buff, strlen(buff), 0,
                     cliaddr, clilen);

/* Check for errors. */
if (count < 0)
    printf("an error occurred\n\r");

```



NU_Recv_From_Raw

```
STATUS NU_Recv_From_Raw(socketd, buff, nbytes,
                        flags, from, addrlen);
```

This function is responsible for receiving data across a network during a Raw IP transfer. It may be called by both the client and the server.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(CHAR *) Pointer to the data buffer. NOTE: because the nbytes parameter is not currently used, the buffer must be large enough to hold the largest datagram that the host expects to receive. In the case of ethernet, this can never be larger than 1500 bytes.
nbytes	(INT16) Specifies the number of bytes of data. NOTE: this parameter is not currently used. Instead the last datagram received is copied as a whole into the location pointed to by the buff parameter.
flags	(INT16) N/A
from	(struct addr_struct *) Pointer to the source protocol-specific address structure.
addrlen	(INT16 *) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, the NU_Recv_From_Raw routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned.



Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NO_DATA_TRANSFER	The data transfer was not completed.

Example

```

INT socketd;    /* the original socket descriptor */
char *buff;     /* pointer to the data buffer */
struct addr_struct cliaddr; /* address we are receiving from */
int clilen; /* length of the client address (unused) */

/* number of bytes successfully transferred */
int count;
.
.
/* Build a socket. */
.
.
/* Allocate memory for client address structure. */
.
.
/* Build the my address structure. */
cliaddr->family = NU_FAMILY_IP; /* Internet Protocol */
cliaddr->port = 0;
cliaddr->id.is_ip_addr[0] = 192;
cliaddr->id.is_ip_addr[1] = 9; /* client machine's */
cliaddr->id.is_ip_addr[2] = 200; /* 4-digit IP number */
cliaddr->id.is_ip_addr[3] = 4;
cliaddr->name = "client_name"

/* Wait for data. */
count = NU_Recv_From (socketd, buff, strlen(buff), 0, cliaddr,
                     clilen);

/* Check for errors. */
if (count < 0)
    printf("an error occurred\n\r");

```



NU_Rip2_Initialize

```
STATUS NU_Rip2_Initialize(ri_ptr, num)
```

This service is called to initialize the RIP2 module. Note that RIP2 is sold as a separate and will not be present unless purchased. RIP2 executes as a task in the system that will periodically update the routing tables. Once initialized no further interaction with RIP2 is required. Invocation of `NU_Rip2_Initialize` is defined as follows:

Parameters

Parameter	Meaning
<code>ri_ptr</code>	(RIP2_STRUCT *) This is a pointer to an array of RIP2 initialization structures. There must be one element in this array for each network interface that should be used with RIP2. This structure is described further below.
<code>num</code>	(INT) This is the number of items in the array pointed to by <code>ri_ptr</code> .

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	RIP2 was initialized successfully.
<code>NU_MEM_ALLOC</code>	A memory allocation failed.
<code>NU_INVALID</code>	A general purpose error condition. Probably indicates that a required resource, task, semaphore, etc. could not be created.



Example

```

VOID RIP2_exec_task(UNSIGNED argc, VOID *argv)
{
    INT          i;
    INT          ret;
    UINT8        ip_addr[] = {192,120,192,30};
    UINT8        subnet[] = {255,255,255,0};
    STATUS        status;
    DEV_DEVICE    devices[1];
    RIP2_STRUCT   rip2[1];

    /* call network initialization */
    ret = NU_Init_Net(&Noncached_Memory);
    if( ret != NU_SUCCESS )
    {
        return;
    }

    /* Access argc and argv just to avoid compilation warnings. */
    status = (STATUS) argc + (STATUS) argv;

    devices[0].dv_name = "PQUICC_0";
    devices[0].dv_hw.ether.dv_irq = 5;
    devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
    devices[0].dv_hw.ether.dv_shared_addr = 0;
    devices[0].dv_init = PQUICC_Init;
    memcpy (devices[0].dv_ip_addr, ip_addr, 4);
    memcpy (devices[0].dv_subnet_mask, subnet, 4);
    devices[0].dv_flags = 0;

    if( (status = DEV_Init_Devices(devices, 1)) != NU_SUCCESS )
    {
        DEMO_Exit(0);
    }

    /* Initialize RIP2 for one device. */

    /* Setup the init structure. */
    rip2[0].rip2_device_name = devices[0].dv_name;
    rip2[0].rip2_metric      = 1;
    rip2[0].rip2_sendmode    = SEND_RIP1;
    rip2[0].rip2_recvmode    = RECV_BOTH;

    /* Call the init function. */
    status = NU_Rip2_Initialize (rip2, 1);

    /* Make sure it worked. */
    if (status != NU_SUCCESS)
        DEMO_Exit(0);
}

```



NU_Select

```
STATUS NU_Select(max_sockets, readfs, writefs, exceptfs, timeout);
```

The `NU_Select` service allows the calling task to check for multiple sockets for data to read, or for acceptable connections. Bit maps are used to indicate which sockets should be checked, each bit corresponding to a specific socket. This service is useful when a task wants to suspend pending an event on multiple sockets, or when a task needs to timeout if the event does not occur by a certain time. The `NU_Recv` and `NU_Recv_From` calls do not allow the caller to specify a timeout value. The caller must choose to whether or not to suspend.

The `NU_Select` service can be used to check for two events, data ready sockets or acceptable connections. Both UDP and TCP sockets can be checked for received data. The `readfs` field is used to specify those sockets that should be checked for received data. Also, both servers and clients can use the `NU_Select` service to check for received data.

`NU_Select` can also be used by TCP servers to check multiple sockets for acceptable connections. Note that the sockets that are checked in this case must be TCP server sockets. That is, they must be “listening” for clients. The `readfs` parameter is also used to indicate those sockets to be checked for acceptable connections.

Parameters

Parameter	Meaning
<code>max_sockets</code>	(INT) Specifies the maximum number of connected socket descriptors.
<code>readfs</code>	(FD_SET *) Pointer to the bitmap of sockets on which the caller wants to check for data.
<code>writefs</code>	(FD_SET *) This parameter is not currently used.
<code>exceptfs</code>	(FD_SET *) This parameter is not currently used.
<code>timeout</code>	(UNSIGNED) Specifies how long to suspend if there is no data on any of the sockets.

There are three options available for the `timeout` parameter:

Parameter	Meaning
<code>NU_NO_SUSPEND</code>	The service returns immediately regardless of whether or not the request can be satisfied .
<code>NU_SUSPEND</code>	The calling task is suspended until at least one of the specified sockets contains data .
<code>timeout value</code>	(1 - 4,294,967,293) The calling task is suspended until data is available, or until the specified number of timer ticks have expired.



On a successful completion, at least one of the specified sockets will be data ready, and a value of `NU_SUCCESS` will be returned. Otherwise, `NU_NO_DATA` is returned. Either `readfs` or `exceptfs` will have one or more bits set. Each socket whose bit is set is either data ready or has an acceptable connection.

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	At least one socket is data ready or a connection is ready to be accepted.
<code>NU_NO_SOCKETS</code>	No sockets were specified in the <code>readfs</code> parameter.
<code>NU_INVALID_SOCKET</code>	No valid sockets were specified in the <code>readfs</code> parameter.
<code>NU_NO_DATA</code>	Indicates whether or not there are any other possible return values.

Example

```
int i;
.
.
.
/* Make sure all bits are initially set to 0 */
NU_FD_Init(&readfs);

/* Set the bit for the first three sockets. */
for (i=0; i<3; i++)
{
    NU_FD_Set(i, &readfs);
}

/* Now call NU_Select to see which, if any, have data. */
if (NU_Select(max_sockets, &readfs, &writefs,
              &exceptfs, NU_NO_SUSPEND)
    == NU_SUCCESS)
{
    /* At least one of the sockets contained received data.
       Check all of them to find those that have data. */
    for (i=0; i<3; i++)
    {
        if (NU_FD_Check(i, &readfs)==NU_FALSE)
            continue;
        .
        .
        .
    }
}
```



NU_Send

```
STATUS NU_Send (socketd, buff, nbytes, flags);
```

This function is responsible for transmitting data across a network during a connection-oriented transfer. It may be called by both the client and the server.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(char *) Pointer to the data buffer.
nbytes	(UINT16) Specifies the number of bytes of data. Valid values for this parameter are in the range 0-IP_MAX_DATA_SIZE(65,495) .
flags	(INT16) N/A

On a successful completion, the NU_Send routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned. The status codes returned and their associated meanings are defined below.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the NU_Socket call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.
NU_NOT_CONNECTED	The data transfer was not completed. This probably occurred because the connection was closed for some reason.
NU_NO_BUFFERS	No transmit buffers were available when the call was made.



Example

```
INT socketd;      /* the original socket descriptor */
char*buff = "x"; /* pointer to the data buffer */
int nbytes = 1;   /* number of bytes to be sent */
int count;        /* number of bytes successfully transferred/
.
.
.
count = NU_Send(socketd, buff, nbytes, 0);

/* if count contains a value greater than or equal to 0, then
count is the number of bytes transferred */
.
.
.
```



NU_Send_To

```
STATUS NU_Send_To (socketd, buff, nbytes, flags, to, addrlen);
```

This function is responsible for transmitting data across a network during a connectionless transfer. It may be called by both the client and the server.

Parameters

Parameters	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(char *) Pointer to the data buffer.
nbytes	(UINT16) Specifies the number of bytes of data.
flags	(INT16) N/A
to	(struct addr_struct *) Pointer to the destination's protocol-specific address structure.
addrlen	(INT16) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, the `NU_Send_To` routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned. The status codes returned and their associated meanings are defined below.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the <code>NU_Socket</code> call.
NU_NO_PORT_NUMBER	No local port number was stored in the socket descriptor.
NU_NO_DATA_TRANSFER	The data transfer was not completed.
NU_INVALID_ADDRESS	The address passed in was most likely incomplete (i.e., missing the IP or port number).



Example

```

INT socketd; /* the original socket descriptor */
char *buff = "x"; /* pointer to the data buffer */
struct addr_struct servaddr; /* address of the remote host */
int servlen; /* length of the server address (unused) */
int count; /* number of bytes successfully transferred */
.
.
.
/* Build a socket.*/
.
.
.
/* Allocate memory for the server structure. */
/* Build the server's address structure. */

servaddr->family = NU_FAMILY_IP; /* Internet Protocol */
servaddr->port = 6000; /* server's port number */
servaddr->id.is_ip_addr[0] = 192;
servaddr->id.is_ip_addr[1] = 9; /* the server machine's */
servaddr->id.is_ip_addr[2] = 200; /* 4-digit IP number */
servaddr->id.is_ip_addr[3] = 1;
servaddr->name = "server_name"

/* Send the data. */
count = NU_Send_To (socketd, buff, strlen(buff), 0, servaddr,
                    servlen);

/* Check the count. */
if (count < 0)
    print("bad error occurred.\n\r");

```



NU_Send_To_Raw

```
STATUS NU_Send_To_Raw (socketd, buff, nbytes, flags, to, addrlen);
```

This function is responsible for transmitting data across a network during a Raw IP transfer. It may be called by both the client and the server.

Parameters

Parameters	Meaning
socketd	(INT) Specifies a socket descriptor.
buff	(char *) Pointer to the data buffer.
nbytes	(UINT16) Specifies the number of bytes of data.
flags	(INT16) N/A
to	(struct addr_struct *) Pointer to the destination's protocol-specific address structure.
addrlen	(INT16) This parameter is reserved for future use. A value of zero should be used.

On a successful completion, the `NU_Send_To_Raw` routine returns the number of bytes transferred. Otherwise, a Nucleus status code is returned. The status codes returned and their associated meanings are defined below.

Return Value

Code	Meaning
NU_INVALID_SOCKET	The socket parameter was not a valid socket value or it had not been previously allocated via the <code>NU_Socket</code> call.
NU_NO_DATA_TRANSFER	The data transfer was not completed.
NU_INVALID_ADDRESS	The address passed in was most likely incomplete (i.e., missing the IP number).



Example

```
INT socketd;                /* original socket descriptor */
char *buff = "x";           /* pointer to the data buffer */
struct addr_struct servaddr; /* address of the remote host */
int servlen;                 /* Unused */
int count;                   /* number of bytes transferred */

/* Build a socket.*/
/* Allocate memory for the server structure. */

/* Build the server's address structure. */
servaddr->family = NU_FAMILY_IP; /* Internet Protocol */
servaddr->port = 0;               /* server's port number */
servaddr->id.is_ip_addr[0] = 192;
servaddr->id.is_ip_addr[1] = 9;  /* the server machine's */
servaddr->id.is_ip_addr[2] = 200; /* 4-digit IP number */
servaddr->id.is_ip_addr[3] = 1;
servaddr->name = "server_name"

/* Send the data. */
count = NU_Send_To_Raw (socketd, buff, strlen(buff), 0,
                        servaddr, servlen);

/* Check the count. */
if (count < 0)
    printf("bad error occurred.\n\r");
```



NU_Setsockopt

```
STATUS NU_Setsockopt(socketd, level, optname, optval, optlen);
```

This function is responsible for setting and clearing socket options for a specified socket. Currently the only supported options are for toggling broadcast and multicast support on a socket. Other options will be supported in the future. The `NU_Setsockopt` service can be used to retrieve socket options.

Parameters

Parameter	Meaning
socketd	(INT) Specifies a socket descriptor.
level	(INT) Specifies the protocol level. The only valid entries for this parameter are <code>SOL_SOCKET</code> , and <code>IPPROTO_IP</code> .
optname	(INT) Specifies an option. Valid options are <code>SO_BROADCAST</code> , <code>IP_ADD_MEMBERSHIP</code> , <code>IP_DROP_MEMBERSHIP</code> , <code>IP_MULTICAST_TTL</code> , and <code>IP_HDRINCL</code> .
optval	(void *) Contains the new value for the option.
optlen	(INT) Specifies the size in bytes of the location pointed to by <code>optval</code> .

Upon successful completion of this service, the option specified by `optname` will be set equal to the value in `optval`. This function also returns a status code. The status codes and their meanings are defined below.

Return Value

Code	Meaning
<code>NU_SUCCESS</code>	Indicates the socket option was set successfully.
<code>NU_INVALID_SOCKET</code>	The specified socket descriptor is invalid.
<code>NU_NOT_CONNECTED</code>	A connection does not exist on the specified socket.
<code>NU_INVALID_LEVEL</code>	The value specified in the <code>level</code> parameter is invalid.
<code>NU_INVALID</code>	Either the <code>optval</code> or <code>optlen</code> parameter has problems.
<code>NU_INVALID_OPTION</code>	The value specified in the <code>optname</code> parameter is invalid.



The following is a list of levels and the associated options that are currently supported by `NU_Setsockopt`.

SOL_SOCKET Level	Options
<code>SO_BROADCAST</code>	Set the broadcast status of a socket. When a socket is created the ability to send broadcasts is enabled by default.

IPPROTO_IP Level	Options
<code>IP_ADD_MEMBERSHIP</code>	Join a multicast group.
<code>IP_DROP_MEMBERSHIP</code>	Leave a multicast group.
<code>IP_HDRINCL</code>	This option, when set, specifies that the application layer is responsible for filling in the IP header. When clear the application will only pass data to the stack, which will then append an IP header. This option is only valid when using the raw IP services.
<code>IP_MULTICAST_TTL</code>	Change the default TTL (Time To Live) for a multicast packets sent on this socket. The default TTL is 1. This keeps routers from forwarding the multicast datagrams beyond the local network.

For an example demonstrating the use of the `NU_Setsockopt` service, please see the `NU_Getsockopt` service call.



NU_Socket

```
INT16 NU_Socket (family, type, protocol);
```

This function is responsible for establishing a new socket descriptor and defining the type of communication protocol to be established. It must be called by both the client and the server whether a connection-oriented or connectionless transfer is being established.

Parameters

Parameter	Meaning
family	(INT16) There exists only one valid entry for the family parameter: <code>NU_FAMILY_IP</code> which specifies Internet Protocol.
type	(INT16) Valid entries for the type parameter include: <code>NU_TYPE_STREAM</code> which specifies TCP Protocol, <code>NU_TYPE_DGRAM</code> , which specifies UDP Protocol, and <code>NU_TYPE_RAW</code> which specifies IP Protocol.
protocol	(INT16) The protocol parameter should be set to <code>NU_NONE</code> . See tables below.

<code>NU_TYPE_STREAM</code> or <code>NU_TYPE_DGRAM</code> type	Protocol
<code>NU_NONE</code>	The protocol field is not used.

<code>NU_TYPE_RAW</code> type	Protocols
<code>IPPROTO_HELLO</code>	HELLO routing protocol
<code>IPPROTO_OSPF</code>	OSPF routing protocol
<code>IPPROTO_RAW</code>	Raw IP protocol
0	Raw IP wildcard protocol

On a successful completion, a socket descriptor is returned from the `NU_Socket` routine with a value greater than or equal to 0. It is used in other routine's parameter lists under the label `socketd`. If the socket call is not successful, a Nucleus status code is returned.





5

Extended Discussion

Blocking on Read Service

Nucleus NET Task Priorities

Including SNMP Support

Domain Name System

IP Multicasting

Dynamic Host Configuration Protocol

Including Routing Information



This chapter provides an extended discussion on several items: Blocking on Read Service, Nucleus NET Task Priorities, Including SNMP support, Domain Name System, Dynamic Host Configuration Protocol, and Including Routing Information Support.

Blocking on Read Service

It is often desirable to determine if data is available from the network without blocking execution of the requesting task. For such cases Nucleus NET provides a mechanism to specify whether or not reads are to be blocked.

This service is provided through the `NU_Fcntl` service call. Via this call, the user specifies if subsequent reads on the indicated socket are to be blocked.

The following code fragment indicates how non-blocking reads can be specified. Blocking reads are the default condition, therefore no service interface is necessary other than the standard `NU_Recv` for a read to be blocked.

```
unsigned int    socketd;          /* socket descriptor      */
unsigned int    count;            /* number of bytes read   */
char           recv_buff[100];    /* buffer for characters read */

/* Get the socket descriptor for the fcntl and read operation */
socketd = socket_descriptor; /* assume from input parameter */

/* Indicate that the subsequent read call is not to block */
NU_Fcntl (socketd, NU_SETFLAG, NU_FALSE, 0);

/* Perform a read from the socket for 100 bytes */
count = NU_Recv (socketd, recv_buff, 100, 0);

/* This socket should block unless specified otherwise. */
NU_Fcntl (socketd, NU_SETFLAG, NU_BLOCK, 0);

/* Check if any characters were read, if so process them. */
if (count > 0)

/* process the data */
```

The blocking capability is only available for TCP services, not UDP or raw reads.

Nucleus NET Task Priorities

Nucleus NET makes use of two tasks. The first is for performing the processing associated with various network events. The second is for de-multiplexing received packets. As shipped, both of these tasks have a priority of 3. This value can be changed by modifying the macros `EV_PRIORITY` and `TM_PRIORITY` in `TARGET.H`. There are no restrictions on the priority of user created application level tasks.



In most cases it is advantageous to have the network tasks equal to or greater than the priority of the other tasks in the system. This is not a requirement, however. In the event that the application level has a higher priority than the network tasks, the user must take steps to insure that the application gives up the thread of execution periodically. Otherwise, the network tasks will be starved for execution time. This is because the lower priority network tasks will be unable to preempt the application.

Including SNMP Support

Nucleus NET can optionally be built to collect the data necessary for your embedded device to be managed by an SNMP Manager. SNMP (Simple Network Management Protocol) defines a framework for the management of network-capable devices. Nucleus SNMP can be purchased separately from Accelerated Technology. Please see the Nucleus SNMP reference manual for more information on SNMP. In the file `ISNMP.H` there is macro that controls the inclusion/exclusion of the SNMP code. The macro is defined as :

```
#define INCLUDE_SNMP    NU_FALSE
```

If Nucleus SNMP is not purchased from Accelerated Technology at the same time as Nucleus NET then SNMP defaults to disabled. To enable SNMP support within Nucleus NET, change the definition of `SNMP_INCLUDED` to:

```
#define INCLUDE_SNMP    NU_TRUE
```

Domain Name System (DNS)

The Domain Name System (DNS) is a distributed database that is used by TCP/IP applications to map between hostnames and IP addresses. DNS defines the protocol that allows clients and servers to communicate with one another. Nucleus NET includes a DNS client, or as it is more commonly known, a DNS resolver. Access to the DNS resolver is through the two Nucleus NET services `NU_Get_Host_By_Name` and `NU_Get_Host_By_Addr`.

When an application accesses either of the two aforementioned services, they first attempt to resolve the information through the local host table. The local host table is defined in the file `HOSTS.C`.

An example host table looks like the following:

```
struct host hostTable[] =
{
    {"ftp.atinucleus.com", 192, 200, 100, 1},
    {"porky_pig", 192, 200, 100, 113},
    {"daffy_duck", 192, 200, 100, 114},
    {"", 0, 0, 0, 0}
};
```



This table maps names to IP addresses. This table should be modified to reflect the hosts that your device will need to communicate with. If the necessary information cannot be resolved locally from the host table, then the DNS resolver will be invoked to resolve the information over the network. In order for the DNS resolver to work there must be a DNS server for it to communicate with, and the IP address of this DNS server must be set. DNS servers can be added and deleted via calls to `NU_Add_DNS_Server` and `NU_Delete_DNS_Server` respectively. The function `NU_Get_DNS_Servers` can be used to retrieve a list of DNS servers that have been registered with Nucleus NET.

The maximum number of DNS servers that can be registered with Nucleus NET is controlled by the following definition, which can be found in at the top of `DNS.H`.

```
#define      DNS_MAX_DNS_SERVERS      5
```

As indicated above `DNS_MAX_DNS_SERVERS` defaults to a value of 5.

DNS is an optional component in that it is not absolutely required for networking. DNS is enabled by default, but it can be turned off. In the file `TARGET.H`, the following definition can be found:

```
#define INCLUDE_DNS      NU_TRUE
```

If `DNS_INCLUDED` is defined to be `NU_FALSE` instead of `NU_TRUE`, then the code to resolve IP addresses and names over the network will not be compiled into the Nucleus NET library. Note that it is still possible to resolve names and addresses through the local host table. You might want to exclude DNS via the network if you are on an isolated network with no DNS server, or if everyone your device will communicate with can be stored in the local host table. Excluding DNS will reduce the size of the Nucleus NET library.

IP Multicasting

IP Multicasting provides a means for a network application to send a single IP datagram to multiple hosts. Multicasting differs from broadcasting in that every host on a network segment receives a broadcast packet. This can lead to unnecessary interrupts for those hosts that are not interested in the broadcast. With multicasting only those hosts that have explicitly joined an IP multicasting group will receive a multicast packet to that address/group. The class D IP addresses define the multicasting IP addresses. These addresses do not define individual interfaces, but instead define groups of interfaces. Hence class D addresses are referred to as groups.

The class D IP addresses are those in the range 224.0.0.0 to 239.255.255.255. Membership in a multicast group is dynamic. An application can join and leave a group on an interface at any time. Nucleus NET applications join or leave a multicast group by utilizing the `NU_Setsockopt` service call. Many IP multicasting groups are reserved for specific applications. RFC 1700 provides a current list of registered groups.



Nucleus NET meets the level 2 (the highest level) host requirements for multicasting. All level 2 conforming hosts are required to join the 224.0.0.1 group at initialization. The 224.0.0.1 or “*all hosts group*” is the group of all hosts on the local subnet. At initialization Nucleus NET will join this group on interfaces that support multicasting.

Multicasting can only be enabled on UDP sockets. This is because UDP is a connectionless protocol. TCP on the other hand establishes a connection with a specific host. Communication over a TCP socket is possible only with that one specific host.

Note that it is possible to exclude support for IP Multicasting by changing the definition of “INCLUDE_IP_MULTICASTING” in the file TARGET.H. By default multicasting support is included in the build of the Nucleus NET library. By turning off support for IP multicasting there will be a slight decrease in executable size.

IGMP (Internet Group Management Protocol) is the means by which IP hosts report their host group memberships to any immediately neighboring multicast routers. Nucleus NET also includes support for IGMP. IGMP is transparent to the user in that there are no API service calls that directly invoke IGMP. Rather each time an application uses `NU_Setsockopt` to join a multicast group, IGMP will be invoked by IP to report the new group membership. Also, if there are any multicast routers on the local network they will periodically send requests for updated group membership information. At this time IGMP will report all group memberships to the router. These requests are sent by routers to the group address of 224.0.0.1, hence the necessity of joining this group during boot up on all interfaces that support multicasting.

DHCP (Dynamic Host Configuration Protocol)

The Dynamic Host Configuration Protocol (DHCP) allows network devices to be assigned an IP address dynamically. DHCP uses the client server model. A DHCP server accepts requests and delivers IP addresses to the client. In addition to IP addresses, other information may be returned by the DHCP server, such as a default gateway, DNS server, etc. DHCP is based on the Bootstrap protocol (BOOTP), adding the capability of automatic allocation of reusable network addresses and additional configuration options. The client assumes that all server offered addresses are not in use. Nucleus NET includes support only for a DHCP client. Access to DHCP is divided into two Nucleus NET services, `NU_Dhcp` and `NU_Dhcp_Release`.

`NU_Dhcp` provides the means by which a Nucleus NET application can discover an IP address. The `NU_Dhcp_Release` service is used to release an IP address obtained from a DHCP server. Please see chapter 4 for more information on each of these service calls.

In the file TARGET.H there is a macro that controls the inclusion/exclusion of the DHCP code in the Nucleus NET library. The macro is defined as follows:

```
#define INCLUDE_DHCP      NU_TRUE
```



By default, NU_DHCP is enabled. To disable NU_DHCP support within Nucleus NET, change the definition of DHCP_INCLUDED to

```
#define INCLUDE_DHCP                NU_FALSE
```

There is one other definition in TARGET.H related to DHCP. This definition controls a function that is used to validate DHCP messages received from a DHCP server. This function, DHCP_Validate, is always called but is stubbed out unless the default definition below is changed. It is up to the user to implement DHCP_Validate to verify the DHCP message is from a valid server.

```
#define DHCP_VALIDATE_CALLBACK      NU_FALSE
```

Note that the DHCP code included with Nucleus NET 4.2 and later is updated from the version included with NET 4.0. If you are upgrading from NET 4.0, please see Appendix C for more details.

Beginning with NET 4.2, the DHCP client will attempt to renew the lease for all IP addresses. If the lease for the original IP can not be renewed, then the device with which the IP address is associated will be shut down when the lease expires. DHCP servers can be configured to give infinite leases, in which case lease renewal is not necessary.

Finally, DHCP tracks the amount of time before a lease expires as timer ticks in a 32-bit integer. With a 10ms clock (100 ticks per second) DHCP can handle a maximum lease time of approximately 248 days. Calculations for leases for longer periods of time can not be made accurately. If the clock rate is faster (more ticks per second) then the maximum lease that can be used effectively will shrink. The lease time is configured through the DHCP server.

For each network device in the system that needs an IP address, the structure DHCP_STRUCT should be created, cleared to zeros, and filled in with the appropriate values. The NU_DHCP_STRUCT is shown below and can be found in DHCP.H. This structure is also used to pass the discovered data back to the application after the NU_Dhcp service has completed.

```
/* The following struct is used to pass in information to the
   NU_Dhcp function call and also used to send information
   back to the caller from the DHCP server. */

struct dhcp_struct {
    UINT8  dhcp_siaddr[4];    /* DHCP server IP address */
    UINT8  dhcp_giaddr[4];    /* Gateway IP address */
    UINT8  dhcp_yiaddr[4];    /* Your IP address */
    UINT8  dhcp_net_mask[4];  /* Net mask for new IP address */
    UINT32 dhcp_xid;          /* Transaction ID. */
    UINT8  dhcp_mac_addr[6];  /* MAC address of client. */
    UINT8  dhcp_sname[64];    /* optional server host name. */
    UINT8  dhcp_file[128];    /* fully pathed filename */
    UINT8  *dhcp_opts;        /* options to discover packet */
                                /* See RFC 2132 for valid options. */
    UINT8  dhcp_opts_len;     /* s length in octets od opts */
};
```



The following shows an example use of NU_Dhcp and NU_Dhcp_Release:

```
#define DEVICE_COUNT    2
NU_DHCP_STRUCT          *dhcp_ptr, *dsp;
NU_DEVICE               devices[DEVICE_COUNT];

void    tcp_server_task(UNSIGNED argc, VOID *argv)
{
    char                c;
    INT16               socketd, newsock;
    struct addr_struct   *servaddr;
    unsigned int         i;
    VOID                *pointer;
    STATUS               status;
    struct addr_struct   client_addr;
    UINT32               ipaddr;
    char                subnet[] = {255,255,255,0};
    STATUS               ret;

    /* These are the dhcp options desired. They are the network
       mask, a list of routers, and a list of Domain Name
       servers/resolvers. */
    UINT8    dhcp_options[] = {DHCP_NETMASK, DHCP_ROUTE,DHCP_DNS};

    /* call network initialization */
    if(NU_Init_Net(&System_Memory) != NU_SUCCESS)
    {
        printf("error at call to NU_Initialize() from
               tcp_server_task.\n");
        NU_Suspend_Task(NU_Current_Task_Pointer());
    }

    /* Setup the information for the first device. */
    devices[0].dv_name = "NE2000_0";
    devices[0].dv_hw.ether.dv_irq = 5;
    devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
    devices[0].dv_hw.ether.dv_shared_addr = 0;
    devices[0].dv_init = NE2000_Init;
    devices[0].dv_flags = 0;
    memcpy(devices[0].dv_ip_addr, "\0\0\0\0", 4);
    memcpy (devices[0].dv_subnet_mask, subnet, 4);

    /* Now setup the second device. */
    devices[1].dv_name = "NE2000_1";
    devices[1].dv_hw.ether.dv_irq = 11;
    devices[1].dv_hw.ether.dv_io_addr = 0x0320L;
    devices[1].dv_hw.ether.dv_shared_addr = 0;
    devices[1].dv_init = NE2000_Init;
    devices[1].dv_flags = 0;
    memcpy(devices[1].dv_ip_addr, "\0\0\0\0", 4);
    memcpy (devices[1].dv_subnet_mask, subnet, 4);

    /* Initialize the devices. */
    NU_Init_Devices(devices, DEVICE_COUNT);
}
```



```

/* allocate memory for the DHCP structures. */
status = NU_Allocate_Memory(&System_Memory, (VOID **)&dhcp_ptr,
                           sizeof(NU_DHCP_STRUCT)
                           * DEVICE_COUNT, NU_NO_SUSPEND);

if( status != NU_SUCCESS )
{
    printf("Can not allocate memory for DHCP_STRUCT.\n");
    DOS_Exit(-1);
}

/* set all DHCP fields to zero value */
memset(dhcp_ptr, 0, sizeof(NU_DHCP_STRUCT) * DEVICE_COUNT );

/* put in user callback function for the vendor options. */
for( i = 0, dsp = dhcp_ptr; i < DEVICE_COUNT; i++, dsp++ )
{
    /* Specify the DHCP options desired. */
    dsp->dhcp_opts = dhcp_options;
    dsp_ptr->dhcp_opts_len = sizeof(dhcp_options);
}

/* dhcp_ptr struct above is left blank unless the caller
   wanted to pass requests to the DHCP server. Get the IP
   address and information from the DHCP Server for the first
   device. */
ret = NU_Dhcp(dhcp_ptr, devices[0].dv_name);
if( ret != NU_SUCCESS )
{
    DOS_Exit(-1);
}

/* Increment the NU_DHCP_STRUCT pointer to point to the next
   devices DHCP struct. */
dhcp_ptr++;

/* Retrieve the DHCP information for the second device. */
ret = NU_Dhcp(dhcp_ptr, devices[1].dv_name);
if( ret != NU_SUCCESS )
{
    DOS_Exit(-1);
}

/* Release the device 0 DHCP obtained device. */
status = NU_Dhcp_Release(dhcp_ptr, devices[0].dv_name);
if( status != NU_SUCCESS )
{
    printf("NU_Dhcp_Release failed.    status = %d\n", status);
}

/* Release the device 1 DHCP obtained device. */
status = NU_Dhcp_Release(dhcp_ptr, devices[1].dv_name);
if( status != NU_SUCCESS )
{
    printf("NU_Dhcp_Release failed.    status = %d\n", status);
}
}

```



Including Routing Information Protocol (RIP2) Support

Nucleus NET can optionally be built to use the routing protocol, known as RIP2 for your embedded device. RIP2 relies on a physical network broadcast to make routing exchanges quickly. RIP2 is used by gateways to periodically exchange their routing database. In this way information on new routes and bad routes can be propagated throughout the network. The message lists each destination along with a distance to the destination. Each gateway will maintain a current routing table based on the RIP2 updates. The routing tables will reflect a change in the topology of the network. Each subsystem must run RIP2 in order for the protocol to reflect every destination in the network.

RIP2 is not included with Nucleus NET. It must be purchased separately. If RIP2 was purchased it will be built into its own library. To take advantage of RIP2 you simply need to link in its library and call the RIP2 initialization function from an application. See `NU_Rip2_Initialize` in Chapter 4 for more information on the initialization of RIP2.

Once initialized the RIP2 task will continue to execute in the background, exchanging routing information with neighboring routers.





6

Using Nucleus NET With Nucleus PLUS

Tasks

Nucleus PLUS Semaphore

Nucleus PLUS Queue



As mentioned previously, Nucleus NET uses various Nucleus PLUS facilities to perform its function. These facilities include two tasks, a resource, and in some cases fixed memory partitions (based on driver construction). The following section describes each of these requirements and the setup required for them to be implemented.

Tasks

Nucleus NET employs two Nucleus PLUS tasks. One task has the responsibility for central coordination and control of the protocol stack and the other can be implemented for either polling or interrupt driven operations.

In addition to the standard tasks used by Nucleus NET, the user interfaces with Nucleus NET from Nucleus PLUS tasks. That is, when a socket service call is made, it is made from within a user task. It is important that these tasks maintain certain parameters to make the Nucleus NET implementation work most efficiently.

When specifying tasks that use Nucleus NET they should:

- Have the same priority as the permanent Nucleus NET tasks (see below)
- Be preemptable

Control/Coordination Task

The control/coordination task for Nucleus NET can be found in the file `EQUEUE.C` and is called `NU_EventsDispatcher`. This task has two primary responsibilities:

- To provide asynchronous activity completion notification (e.g., connection complete, data in receive buffer).
- To provide timing services (e.g., retransmission timers for TCP).

The `NU_EventsDispatcher` task is created with a priority defined by the macro `EV_PRIORITY`. `EV_PRIORITY` is defined in the file `CONFIG.H`. The priority for the Nucleus NET tasks should be commensurate with the importance of the networking support required in any application. If the networking is critical, its priority should be high. As the networking support decreases in importance, its priority should be lower.

The `NU_EventsDispatcher` task waits on a Nucleus PLUS queue (`eQueue`) for its information. This queue is used internally by Nucleus NET, so the user should never have to access or modify this queue.



Timer Notification Task

This task's implementation will differ based on the type of driver being employed. If the driver is interrupt driven, this task simply obtains the Nucleus NET semaphore and calls `demux`. In the interrupt driven version, the `timer_task` is activated from an interrupt service routine. If the driver is polled, this task should periodically call `netsleep` after obtaining the Nucleus NET semaphore. After releasing the semaphore, the `timer_task` should call `NU_Sleep` for at least one clock tick.

The `timer_task` is created with a priority defined by the macro `TM_PRIORITY`. `TM_PRIORITY` is defined in the file `CONFIG.H`. The priority for the Nucleus NET `timer_task` should be commensurate with the importance of the networking support required in any application. If the networking is critical, its priority should be high. As the networking support decreases in importance, its priority should be lower.

Note that the mode in which the `timer_task` operates (polled or interrupt) is determined at compile time. If `INTERRUPT` is defined at compile time then the interrupt version will be used. Otherwise the polled version is used.

Notice that as most Nucleus PLUS tasks, `timer_task` operates in an infinite loop. After entering that loop it acquires the Nucleus PLUS Semaphore (`TCP_Resource`) and will suspend indefinitely until it can get the semaphore.

In the polling version, when the semaphore is acquired it makes a call to `netsleep`. This routine will then call the `demux` routine and process all the currently stored packets. Each packet will be processed based on the type code and then the buffers will be updated to reflect that the packet has been processed. When all packets have been received, `timer_task` regains control from `netsleep` and releases the semaphore. After releasing the semaphore, `timer_task` sleeps for a period of one clock tick. When it wakes up it will begin the checking process all over again.

In the interrupt driven version, when the semaphore is obtained it makes a call to `demux`. This routine will process all the currently stored packets. Each packet will be processed based on the type code and then the buffers will be updated to reflect that the packet has been processed. When all packets have been received, `timer_task` regains control from `demux` and releases the semaphore. After releasing the semaphore, `timer_task` suspends until activated by an interrupt service routine. When it resumes `timer_task` will begin the checking process all over again.

The `demux` routine uses the global variable `buffer_list` to see if there are any packets that need to be processed. This variable contains a pointer to the head of a buffer list. Each buffer in the list contains one packet. If the head pointer is `NULL` then the list is currently empty. The driver is responsible for placing packet buffers onto this list.



Nucleus PLUS Semaphore

In order to guard against re-entrancy problems, Nucleus NET employs a Nucleus PLUS semaphore (a counting semaphore). This semaphore is actually declared as `TCP_Resource` so as to be name compatible with the RTX version of the generic code. This semaphore is used internally by Nucleus NET, and therefore should not be used or changed by the user.

Nucleus PLUS Queue

As mentioned above, the `NU_EventsDispatcher` task receives notification of events via a Nucleus PLUS queue. As shipped, the software defines the queue as having 100 (`SOCKETQUEUE_SIZE`) items of (`SOCKETQUEUE_ELEMENTS`) in length. If the system experiences a backlog of queue items (e.g., when queues are posted by `netputevent`, it suspends because the queue is full) the queue size should be increased. These constants are defined in the file `CONFIG.H`.



7

Driver Assistance Functions

Introduction

"MII_AutoNeg" (MI Automatic
Negotiate)



Introduction

This chapter describes services that are supplied by Nucleus NET in order to help speed device driver development. These services are considered generic across network drivers. Therefore any driver that requires the specified type of functionality can use them.

“MII_AutoNeg” (MII Automatic Negotiate)

This function assists in the process of setting up a link between a 100 Mbps Ethernet device and the link partner. To enable auto negotiation simply call this function from the appropriate place in the Ethernet driver during chip initialization.

Invocation of the MII *Automatic Negotiate* routine is defined as follows:

```
STATUS MII_AutoNeg(deviceP, phyAddr, retries, isFullDuplexP,
                  is100MbpsP, MiiRead, miiWrite)
```

The following two routines are supplied by the driver to the MII_AutoNeg function. They are used to read from and write to the MII registers and must be correct in for the auto negotiation to be successful.

Invocation of the MII *read* routine is defined as follows:

```
typedef STATUS (*mii_ReadMII)(DV_DEVICE_ENTRY* deviceP, int phyAddr,
                             int regAddr, unsigned short* inP);
```

Invocation of the MII *write* routine is defined as follows:

```
typedef STATUS(*mii_WriteMII)(DV_DEVICE_ENTRY* deviceP, int phyAddr,
                              int regAddr, unsigned short out);
```



Parameters

Parameter	Meaning
deviceP	(DV_DEVICE_ENTRY*) pointer to DV_DEVICE_ENTRY for the device, to pass to the supplied mii_ReadMII and mii_WriteMII routines.
phyAddr	(INT) PHY address on MII bus .
retries	(UINT32) number of times to poll for auto-negotiation to complete, or zero to poll indefinitely.
isFullDuplexP	(INT*) pointer to boolean indicating if link can be full duplex.
is100Mbps	(INT*) pointer to boolean indicating if link can be 100Mbps.
MiiRead	(mii_ReadMII) Driver supplied MII Read routine.
miiWrite	(mii_WriteMII) Driver supplied MII Write routine.

Return Value

Code	Meaning
NU_SUCCESS	The PHY auto-negotiated successfully with the link partner, and the results are in *isFullDuplexP and *is100MbpsP.
NU_TIMEOUT	The specified number of retries elapsed without auto-negotiation completing.
NU_NOT_PRESENT	The supplied phyAddr was not valid
NU_INVALID_OPERATION	*isFullDuplexP or *is100MbpsP were not both either NU_TRUE or NU_FALSE
NU_INVALID_FUNCTION	miiRead or miiWrite was NU_NULL
NU_INVALID_POINTER	deviceP, isFullDuplexP, or is100Mbps was NU_NULL

Example

```

int isFullDuplex;
int is100Mbps;
STATUS status = NU_SUCCESS;
.
.
.
status = MII_AutoNeg(deviceP, FADS_LXT970_PHY_ADDR,
                    FEC_AUTONEG_TRIES, &isFullDuplex,
                    &is100Mbps, ReadMII, WriteMII);

```

The "MII_AutoNeg" call uses the functions ReadMII, and WriteMII, which should be included with the driver.







Appendix

Values Returned by
Socket Interface



The following #defines and their associated values are error codes returned to applications using the socket interface to Nucleus NET.

Symbol	Value	Description
NU_ARP_FAILED	-26	ARP failed to resolve the hardware address.
NU_INVALID_PROTOCOL	-27	Invalid network protocol.
NU_NO_DATA_TRANSFER	-28	Data was not written/read during send/receive function.
NU_NO_PORT_NUMBER	-29	No local port number was stored in the socket descriptor.
NU_NO_TASK_MATCH	-30	No task/port number combination existed in the task table.
NU_NO_SOCKET_SPACE	-31	The socket structure list was full when a new socket descriptor was requested.
NU_NO_ACTION	-32	No action was processed by the function.
NU_NOT_CONNECTED	-33	A connection has been closed by the network
NU_INVALID_SOCKET	-34	The socket ID passed in was not in a valid range.
NU_NO SOCK_MEMORY	-35	Memory allocation failed for internal sockets structure.
NU_NOT_A_TASK	-36	Attempt was made to make a sockets call from an interrupt without doing context save.
NU_INVALID_ADDRESS	-37	An incomplete address was sent up to a UDP call.
NU_NO_HOST_NAME	-38	No host name specified in a connect call where a machine was not previously set up.
NU_RARP_INIT_FAILED	-39	During initialization RARP failed.
NU_BOOTP_INIT_FAILED	-40	During initialization BOOTP failed.
NU_INVALID_PORT	-41	The port number passed in was not in a valid range.
NU_NO_BUFFERS	-42	There were no buffers to place the outgoing packet in.
NU_NOT_ESTAB	-43	A connection is open, but not in an established state.
NU_INVALID_BUF_PTR	-44	The buffer pointer is invalid.
NU_WINDOW_FULL	-45	The foreign host's in window is full.
NU_NO_SOCKETS	-46	No sockets were specified.
NU_NO_DATA	-47	None of the specified sockets were data ready.
NU_INVALID_LEVEL	-48	The specified level is invalid.



Appendix A - Values Returned By Socket Interface

Symbol	Value	Description
NU_INVALID_OPTION	-49	The specified option is invalid.
NU_INVAL	-50	General purpose error condition.
NU_ACCESS	-51	The attempted operation is not allowed on the socket.
NU_ADDRINUSE	-52	The IP Multicast membership already exists.
NU_HOST_UNREACHABLE	-53	Host unreachable.
NU_MSGSIZE	-54	Packet is too large for interface.
NU_NOBUFS	-55	Could not allocate a memory buffer.
NU_UNRESOLVED_ADDR	-56	The MAC address was not resolved.
NU_CLOSING	-57	The other side in a TCP connection has sent a FIN.
NU_MEM_ALLOC	-58	Failed to allocate memory.
NU_RESET	-59	A multicast membership was added and the MAC chip needs to be reset. This is not passed up to the application level.
NU_INVALID_LABEL	-60	Indicates a domain name with an invalid label.
NU_FAILED_QUERY	-61	No response received for a DNS Query.
NU_DNS_ERROR	-62	A general DNS error status.
NU_NOT_A_HOST	-63	The host name was not found.
NU_INVALID_PARM	-64	A parameter has an invalid value.
NU_NO_IP	-65	An IP address was not specified for the device.
NU_DHCP_INIT_FAILED	-66	During initialization DHCP failed.
NU_DHCP_REQUEST_FAILED	-67	DHCP could not successfully complete.
NU_BOOTP_SEND_FAILED	-68	BOOTP could not send a request.
NU_BOOTP_RECV_FAILED	-69	BOOTP could not make the NU_Recv_From call to get the BOOTP server's response.
NU_BOOTP_ATTACH_IP_FAILED	-70	BOOTP failed trying to attach the IP address to the network device.
NU_BOOTP_SELECT_FAILED	-71	BOOTP could not make the NU_Select call to wait for the response from a BOOTP server.
NU_BOOTP_FAILED	-72	General BOOTP failure. Possibly because no BOOTP server was available or configure properly.



B

Appendix

Sample Application

This is an example of a simple Nucleus NET application. The demo is an echo client. The client application demonstrates the initialization of Nucleus NET, the registering of a Physical device with a stack, and the use of DHCP to acquire an IP address. Then, once an IP address has been acquired, the client will establish a connection with a TCP echo server, bounce some packets off of the server, and then repeat the process again for some number of iterations.

```
#include "externs.h"
#include "socketd.h"      /* socket interface structures */
#include "tcpdefs.h"
#include "target.h"
#include "nucleus.h"
#include "dhcp.h"

/***** Platform Specific Values *****/
#define DEVICE_NAME      "PQUICC_0"
#define DEVICE_IRQ       5
#define DEVICE_IO_ADDR   0x0300L
#define DEVICE_INIT_FUNCTION PQUICC_Init

extern STATUS PQUICC_Init(DV_DEVICE_ENTRY *device);
/*****End Platform Specific Values *****/

#define NUMBER_OF_ITERATIONS      1
#define ECHO_SERVER                "www.example.net"

#define printf PRINTF

/* Define Application data structures. */
NU_MEMORY_POOL      System_Memory;
NU_TASK             tcp_client_task_ptr;

/* Define prototypes for function references. */
void TCP_client_task(UNSIGNED argc, VOID *argv);
void PRINTF(char*, ...);
void DEMO_Exit(int);

/*****
*
* FUNCTION
*
*      Application_Initialize
*
* DESCRIPTION
*
*      Define the Application_Initialize routine that determines the
*      initial Nucleus PLUS application environment.
*
*****/

void Application_Initialize (void *first_available_memory)
{
    VOID *pointer;
    STATUS status;

    /* Create a system memory pool that will be used to allocate task
       stacks, queue areas, etc. */
    status = NU_Create_Memory_Pool(&System_Memory, "SYSMEM",
                                   first_available_memory, 400000, 50,
                                   NU_FIFO);
}
```



```

if (status != NU_SUCCESS)
{
    printf ("Can not create the System Memory Pool.\n");
    DEMO_Exit (-1);
}

/* Create each task in the system. */

/* Create TCP client task. */
status = NU_Allocate_Memory (&System_Memory, &pointer, 5000,
                             NU_NO_SUSPEND);
if (status != NU_SUCCESS)
{
    printf ("Can not create memory for tcp_client_task.\n");
    DEMO_Exit (-1);
}
status = NU_Create_Task (&tcp_client_task_ptr, "TCPCLI",
                        TCP_client_task, 0, NU_NULL, pointer, 5000,
                        3, 0, NU_PREEMPT, NU_START);
if (status != NU_SUCCESS)
{
    printf ("Cannot create TCP_client_task\n\r");
    DEMO_Exit (2);
}

} /* end Application_Initialize */

/*****
*
* FUNCTION
*
*      TCP_client_task
*
* DESCRIPTION
*
* This function is the entry point for a function that initializes
* Nucleus NET, registers one physical interface/device with the stack,
* and invokes DHCP to acquire an IP address. The task is in a loop such
* that it will repeatedly obtain an IP address, bounce some packets off
* of an ECHO server, release the IP address, and then repeat the whole
* process again.
*
*****/
void TCP_client_task(UNSIGNED argc, VOID *argv)
{
    int socketd; /* the socket descriptor */
    struct addr_struct servaddr; /* holds the server address structure */
    unsigned int *return_ptr;
    STATUS status;
    int bytes_received;
    int bytes_sent;
    char *buffer;
    INT16 servlen;
    NU_HOSTENT hentry;
    NU_DEVICE devices[1];
    unsigned long total_pkts = 0;
    INT32 i,j;
    DHCP_STRUCT *dhcp_ptr;

```



```

/* These are the dhcp options desired. They are the network mask, a
   list of routers, and a list of Domain Name servers/resolvers. */
UINT8 dhcp_options[] = {DHCP_NETMASK, DHCP_ROUTE, DHCP_DNS};

/* call network initialization */
if(NU_Init_Net(&System_Memory) != NU_SUCCESS)
{
    printf("error at call to NU_Initialize() from TCP_client_task");
    DEMO_Exit(1);
}

/* Set up the device initialization function. In this example an IP
   address of 0.0.0.0 is used. So the device will not be able to send
   or receive data until an IP address is attached. In this Example DHCP
   will be used below to acquire an IP address. */
devices[0].dv_name = DEVICE_NAME;
devices[0].dv_hw.ether.dv_irq = DEVICE_IRQ;
devices[0].dv_hw.ether.dv_io_addr = DEVICE_IO_ADDR;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = DEVICE_INIT_FUNCTION;
devices[0].dv_flags = 0;
memcpy(devices[0].dv_ip_addr, "\0\0\0\0", 4);
memcpy(devices[0].dv_subnet_mask, "\0\0\0\0", 4);

/* Initialize the device. */
NU_Init_Devices(devices, 1);

/* allocate memory for the DHCP structures. */
status = NU_Allocate_Memory(&System_Memory, (VOID **)&dhcp_ptr,
                           sizeof(DHCP_STRUCT),
                           NU_NO_SUSPEND);

if( status != NU_SUCCESS )
{
    printf("Can not allocate memory for DHCP_STRUCT.\n");
    DEMO_Exit(-1);
}

/* Allocate space for the buffer. */
status = NU_Allocate_Memory (&System_Memory, (void *) &buffer, 2000,
                             NU_SUSPEND);

if (status != NU_SUCCESS)
{
    /* Can't allocate memory, get out. */
    printf("Cannot allocate memory for TCP buffer\n\r");
    DEMO_Exit(2);
}

for (j = 0; j < NUMBER_OF_ITERATIONS; j++)
{
    /* set all DHCP fields to zero value */
    memset(dhcp_ptr, 0, sizeof(DHCP_STRUCT));
}

```



```

/* put in user callback function for the vendor options. */
#ifdef(DHCP_VALIDATE_CALLBACK)
    /* prototype is defined in dhcp.h */
    dhcp_ptr->dhcp_valfunc = DHCP_Validate;
#endif

/* Specify the DHCP options desired. */
dhcp_ptr->dhcp_opts = dhcp_options;
dhcp_ptr->dhcp_opts_len = sizeof(dhcp_options);

/* dhcp_ptr struct above is left blank unless the caller wanted to pass */
/* requests to the DHCP server. */
status = NU_Dhcp(dhcp_ptr, DEVICE_NAME);

if( status != NU_SUCCESS)
{
    continue;
}

/* open a connection via the socket interface */
if ((socketd = NU_Socket(NU_FAMILY_IP, NU_TYPE_STREAM, 0)) < 0)
{
    DEMO_Exit(-1);
}

/* If unable to have DNS operational fill out the hosts.c
   with the desired server name and ip address */
if (NU_Get_Host_By_Name (ECHO_SERVER, &hentry) != NU_SUCCESS)
    DEMO_Exit(3);

/* fill in a structure with the server address */
servaddr.family = NU_FAMILY_IP;
memcpy(&servaddr.id, hentry.h_addr, hentry.h_length);
servaddr.port = 7;
servaddr.name = "servername";

/* Connect to the server. */
if ((NU_Connect (socketd, &servaddr, 0)) < 0 )
{
    DEMO_Exit(-1);
}

/* Bounce some packets off of the echo server. */
for (i = 0; i < 10; i++)
{
    sprintf(buffer, "This is TCP test packet # %lu", total_pkts);

    /* Send the datagram. */
    bytes_sent = NU_Send(socketd, buffer, 50, 0);

    /* If the data was not sent, we have a problem. */
    if (bytes_sent < 0)
    {
        printf("\n\rError in sending to the TCP server\n\r");
        DEMO_Exit(4);
    }
}

```



```

/* Only wait for a string if we sent something. */
if (bytes_sent > 0)
{
    /* turn on the "block during a read" flag */
    NU_Fcntl(socketd, NU_SETFLAG, NU_BLOCK);

    /* Go get the server's response. */
    bytes_received = NU_Recv(socketd, buffer, bytes_sent, 0);

    /* turn off the "block during a read" flag -
       other reads may not want to block */
    NU_Fcntl(socketd, NU_SETFLAG, NU_FALSE);

    /* If we got an error, its bad. */
    if (bytes_received < 0)
    {
        printf("\nError in receiving from TCP server\n");
        DEMO_Exit(5);
    }

    /* NULL terminate the string. */
    buffer[bytes_received] = (char) 0;

    /* Show the user that we got something back. */
    printf("\n\r The string received was: %s\n\r", buffer);
} /* if bytes_sent > 0 */

total_pkts++;

} /* while not quitting */

/* close the connection */
if ((NU_Close_Socket(socketd)) != NU_SUCCESS)
{
    printf("\nError from NU_Close_Socket.");
}

/* Give the close time to complete before releasing the IP address. */
NU_Sleep(50);

/* Call DHCP to release the IP address. */
status = NU_Dhcp_Release(dhcp_ptr, "PQUICC_0");

if( status != NU_SUCCESS )
{
    printf("NU_Dhcp_Release failed.    status = %d\n", status);
    DEMO_Exit(0);
}

NU_Sleep(2);

/* Indicate that all went well. */
printf("Successful completion of TCP Client program\n\r");
DEMO_Exit(0);
}

```



```
void DEMO_Exit(int a)
{
    while(1);
}

void PRINTF(char* string, ...)
{
}
```







Appendix

Changes to the Nucleus NET API



While Nucleus NET 4.0 contains many improvements over the prior versions of Nucleus NET, an attempt was made to modify the existing API as little as possible. This section describes the changes that were necessary.

New Initialization Sequence

The initialization of Nucleus NET was changed. Prior to NET 4.0, the application made a single service call to `NU_Init_Net` to perform initialization. Now, two service calls are required, `NU_Init_Net` and `NU_Init_Devices`. `NU_Init_Net` no longer accepts any parameters and only initializes the stack. `NU_Init_Devices` is now responsible for initializing MAC layer devices. Please see those respective calls in Chapter 4 for more information.

The following is a typical example of the new installation procedure. Note that `NU_Init_Net` must be called before `NU_Init_Devices` and both must be called before any other network services. This example shows the initialization of a single MAC layer device, in this case an NE2000 Ethernet controller. However, `NU_Init_Devices` can be used to initialize multiple MAC layer devices.

```
UINT8          ip_addr[] = {206,202,34,91};
UINT8          subnet[] = {255,255,255,0};
DEV_DEVICE     devices[1];
.
.
.
/* Initialize the protocol stack. */
if(NU_Init_Net() != NU_SUCCESS)
{
    printf("error at call to NU_Initialize() from
           tcp_server_task.\n");
    NU_Suspend_Task(NU_Current_Task_Pointer());
}

/* CLear the device array. */
UTL_Zero(devices, sizeof(*devices));

devices[0].dv_name = "NE2000_0";
devices[0].dv_hw.ether.dv_irq = 5;
devices[0].dv_hw.ether.dv_io_addr = 0x0300L;
devices[0].dv_hw.ether.dv_shared_addr = 0;
devices[0].dv_init = NE2000_Init;
devices[0].dv_flags = 0;
memcpy (devices[0].dv_ip_addr, ip_addr, 4);
memcpy (devices[0].dv_subnet_mask, subnet, 4);
memcpy (devices[0].dv_gw, gw_addr, 4);

DEV_Init_Devices(devices, 1);
```



NU_Fcntl

Prior to NET 4.0 this service was called `NU_fcntl`. The `F` in `fcntl` was capitalized to be consistent with the rest of the Nucleus NET service calls. The last parameter in the `NU_Fcntl` service was dropped. See the next change for details.

Optimized Data Handling

Prior to NET 4.0 Nucleus NET included support for ODH or optimized data handling. ODH eliminated a copy on both transmits and receives, making those more efficient. NET 4.0 no longer stores packets in a contiguous memory area but instead chains smaller memory buffers together to form a large packet. This data chaining makes more efficient use of available memory but renders the ODH support self-defeating. This is because with the data chained across the multiple buffers, either the application would have to know about the way Nucleus NET stores packets in these chains, or the packet would have to be placed into a contiguous memory area before passing it up to the application. This would defeat the purpose (eliminating a copy) of having ODH in the first place. The above means that the following service calls no longer exist:

```
NU_Allocate_Buffer
NU_Deallocate_Buffer
NU_ODH_Recv
```

Also, the last parameter in the `NU_Fcntl` service call was dropped. The last parameter was used to indicate if support for ODH was desired on a socket.

Unfortunately this means that applications that made use of ODH will have to be modified to no longer use those calls.

More options were added to the `NU_Getsockopt` and `NU_Setsockopt` services. All of the added options are related to IP multicasting.

New API Service Calls

The following service calls were added:

```
NU_Add_Route
NU_Init_Devices
NU_Abort
NU_Ioctl
NU_Dhcp
NU_Dhcp_Release
NU_Is_Connected
```



NU_Init_Net

The `NU_Init_Net` service call gained a parameter in NET 4.2. This parameter can be used to specify a non-cached memory pool from which the Nucleus NET buffers can be allocated. It might also be useful for drivers to allocate from this non-cached memory pool. This is because on architectures where the `System_Memory` pool is in cached memory and the hardware device has the ability to read and write RAM. Cache flushes are required when the driver software accesses buffers or data structures that are shared with hardware. The flushes are expensive. A better solution is to put these data structures in non-cached memory. For systems that do not need to specify a non-cached memory pool, the `System_Memory` pool can be used here. The new macro `NET_VERSION_COMP`, described below, can be used to maintain compatibility with existing Nucleus NET applications. See below for more details.

A New Standard Set of Data Types

Nucleus NET will now use a single set of data types. In the past, Nucleus NET had a couple of different typedefs for a unsigned long int, unsigned short int, etc. A new standard set of types is now used in the stack and will be phased in throughout the Nucleus product line. The new types, like the old, can be found in `TARGET.H`. These types follow.

```
typedef char          INT8;
typedef unsigned char UINT8;
typedef signed short  INT16;
typedef unsigned short  UINT16;
typedef signed long   INT32;
typedef unsigned long  UINT32;
```

Raw IP Support was Added

Support for was added for a Raw IP interface. This interface allows an application to bypass the TCP and UDP layers and transmit IP packets directly. Please see the `NU_Recv_From_Raw` service call, the `NU_Send_To_Raw` service call, and Chapter 3 for more information on the raw IP interface.



Ping Request

Nucleus NET has always had the ability to respond to PING requests. However, NET 4.2 is the first version that includes a service that allows an application to generate a PING request. Utilizing this feature a NET 4.2 application can PING foreign hosts to see if they are still alive. The new API function `NU_Ping` was added for this purpose.

No Longer Necessary to Align Received Data

In the past Nucleus NET drivers that were used on architectures that required strict data byte alignment were required to align received data on a half word boundary. This is no longer necessary. Now received data can be located at any boundary.

DHCP Module Upgraded

The DHCP client support has been upgraded. The DHCP client can now renew the lease. The client will automatically check for the vendor options that specify a network mask, DNS server address, and default gateway address. If any of those options are present they will be processed by DHCP, i.e., the application does not have to add a route to the default gateway, or register the DNS server with Nucleus NET. The only remaining limitation is the inability to acquire a new IP address if the lease for the current one expires, either because the server will not renew it or the server has gone down.

Multiple DNS Server Support

Nucleus NET 4.2 includes support for multiple DNS servers. Also, added were three new API services for adding, deleting, and retrieving a list of DNS servers. The new services are `NU_Add_DNS_Server`, `NU_Delete_DNS_Server`, and `NU_Get_DNS_Servers` respectively.

New API Service Calls

The following new service calls have been added:

```
NU_Add_DNS_Server
NU_Delete_DNS_Server
NU_Get_DNS_Servers
NU_Ping
NU_Recv_From_Raw
NU_Send_To_Raw
```



